DOCUMENT RESUME

ED 207 583 IR 009 701

AUTHOR	Goldstein, Ira P.; Miller, Mark L.
TITLE	Structured Planning and Debugging: A Linguistic Theory of Design. AI Memo 387.
INSTITUTION	Massachusetts Inst. of Tech., Cambridge. Artificial Intelligence Lab.
SPONS AGENCY	Advanced Research Projects Agency (DOD), Washington, D.C.; National Science Foundation, Washington, D.C.
REPORT NO	LOGO-34
PUB DATE	Dec 76
GRANT	NSF-EC40708X; ONR-N00014-75-C-0643
NOTE	92p.; For related documents, see IR 009 697, IR 007 700, and IR 009 702.
EDRS PRICE	MF01/PC04 Plus Postage.
DESCRIPTORS	Artificial Intelligence; Computer Graphics; *Computer Programs; Diagrams; *Linguistic Theory; *Planning; *Problem Solving; Programming; Task Analysis
IDENTIFIERS	Debugging Aids; *Structured Planning and Debugging Editor

ABSTRACT

A unified theory of planning and debugging is explored by designing a problem solving program called PATN. PATN uses an augmented transition network (ATN) to represent a broad range of planning techniques, including identification, decomposition, and reformulation. (The ATN is a simple yet powerful formalism which has been effectively utilized in computational linguistics.) PATN's plans may manifest "rational bugs," which result from heuristically justifiable but incorrect arc transitions in the planning ATN. This aspect of the theory is developed by designing a complementary debugging module called DAPR, which would diagnose and repair the errors in PATN's annotated plans. The investigation is incomplete as PATN has not yet been implemented, but sufficient detail is presented to provide a theoretical framework for reconceptualizing Sussman's HACKER research. Since a detailed study of planning and debugging techniques is a prerequisite for complete fulfillment of Dijkstra's objectives of program reliability, readability, portability, and so on, the theory is called "Structured Planning and Debugging" to emphasize its potential role in this enterprise. Fifty-seven references are listed. (Author/LLS)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED207583

U S DEPARTMENT OF HEALTH, EDUCATION & WELFARE NATIONAL INSTITUTE OF EDUCATION

THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL NATIONAL INSTITUTE OF EDUCATION POSITION OR POLICY

IR

Massachusetts Institute of Technology
Artificial Intelligence Laboratory

AI Memo 387 December 1976 Logo Memo 34

Structured Planning and Debugging

A Linguistic Theory of Design

Ira P. Goldstein and Mark L. Miller

A unified theory of planning and debugging is explored by designing a problem solving program called PATN. PATN uses an augmented transition network (ATN) to represent a broad range of planning techniques, including identification, decomposition, and reformulation. (The ATN [Woods 1970] is a simple yet powerful formalism which has been effectively utilized in computational linguistics.)

PATN's plans may manifest "rational bugs," which result from heuristically justifiable but incorrect arc transitions in the planning ATN. This aspect of the theory is developed by designing a complementary debugging module called DAPR, which would diagnose and repair the errors in PATN's annotated plans.

The investigation is incomplete: PATN has not yet been implemented. But sufficient detail is presented to provide a theoretical framework for reconceptualizing Sussman's [1973] HACKER research.

Since a detailed study of planning and debugging techniques is a prerequisite for complete fulfillment of Dijkstra's objectives of program reliability, readability, portability, and so on, the theory is called, "Structured Planning and Debugging," to emphasize its potential role in this enterprise.

This report is a revised version of AI Working Paper 125 (Logo Working Paper 55). It describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. It was supported in part by the National Science Foundation under grant C40708X, in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643, and in part by the Division for Study and Research in Education, Massachusetts Institute of Technology.

IR009701

Contents

1. Introduction

2. Structured Planning
   - 2.1. An Augmented Transition Network for Planning
   - 2.2. PATN's Plan Node
   - 2.3. Problem Identification
   - 2.4. Problem Decomposition
   - 2.5. Decomposition by Conjunction
   - 2.6. PATN's Subgraph for Conjunction
   - 2.7. Composition by Sequential Refinement
   - 2.8. Decomposition by Repetition
   - 2.9. Problem Reformulation

3. Searching for the Plan
   - 3.1. Lookahead
   - 3.2. Least Commitment
   - 3.3. Differential Diagnosis
   - 3.4. Lemma Libraries

4. Structured Debugging
   - 4.1. Model Diagnosis
   - 4.2. Process Diagnosis
   - 4.3. Plan Diagnosis
   - 4.4. Repair
   - 4.5. Limitations of the ATN Theory of Bugs

5. Reconceptualizing HACKER
   - 5.1. Bugs Arising from Incomplete Plans
   - 5.2. Bugs Arising from Incorrect Conjunctive Plans
   - 5.3. Bugs Arising from Incorrect Disjunctive Plans
   - 5.4. Generalizing the HACKER Paradigm

6. Conclusions
   - 6.1. Limitations and Extensions of Structured Planning
   - 6.2. Summary of the Structured Debugging Viewpoint
   - 6.3. Protocol Analysis
   - 6.4. Structured Programming
   - 6.5. AI-based Computer Aided Instruction
   - 6.6. The Science of Heuristic

7. Notes
8. References

Thanks are due to M. Genesereth, B. Kuipers, D. Marr, D. McDermott and S. Rosenberg, for carefully criticizing an earlier version of this paper; and to Carol Roberts, for help with the illustrations.

1. Introduction

> Though it is difficult to prescribe any Thing in these Sorts of Cases, and every Person's own Genius ought to be his Guide in these Operations; yet I will endeavor to show the Way to Learners.
>
> Newton, Universal Arithmetick (translated by Ralphson, 1769, p. 198), from [Polya 1965, p. 89].

The structured programming movement [Dahl et al. 1972] has focused the concern of computer scientists on the process of creating programs. Work in artificial intelligence (AI) has developed a complementary theory of debugging [Sussman 1973; Goldstein 1974]. But, except for Sacerdoti's [1975] work on *procedural nets*, a comprehensive approach has not yet been attempted. This is a preliminary report on a theory called *Structured Planning and Debugging*[1] which we believe to be a step towards an integrated theory of design.

Our task has two aspects. First, we hope to understand certain intricacies of planning and debugging, such as are encountered in the design of programs which must take into account interactions in achieving dependent subgoals. The second aspect of our task is to seek a representational framework in which to elucidate these subtleties, and in which to structure a wide variety of planning techniques. Our methodology is to begin with simple but clear formalisms, studying their virtues and limitations. Our research plan is then to investigate a series of progressively more powerful and elaborate representations, after we have reached a solid understanding as to where the extra power is needed, and why.

In earlier work, we have studied one particularly simple representation for planning knowledge: context free grammars (e.g., [Miller & Goldstein 1976b]). In this essay we pursue the investigation by exploring the use of a more elaborate formalism. We utilize an *augmented transition network* (ATN) [Woods 1970][2] to represent an hierarchical taxonomy of planning methods. We are exploring ATNs as a representation for planning concepts because they directly generalize context free grammars, and because work in computational linguistics has shown them

to be both perspicuous and rich in expressive power.[3]

An ATN is a *finite state transition graph* with labelled states and arcs, augmented by recursion and a finite number of registers. Associated with each arc may be conditions on following the arc, and actions to be executed if the arc is followed. Typically the conditions are restricted to Boolean predicates over the contents of the registers. The actions are restricted to structure building and modifying the contents of the registers.

We apply the ATN formalism to planning by representing possible planning decisions as transitions between nodes of the network. The semantic context, including the problem description, is defined using the ATN's registers. Pragmatic knowledge, specifying which planning strategies to apply in which situations, is modeled by arc transition constraints. The ATN constructed in this fashion defines a problem solving program called PATN (Planning ATN).

> CAVEAT: to simplify the discussion we speak of PATN as if it were a working program. However, at this point in time, PATN is only a design. This design is sufficiently precise to be hand-simulated on simple problems, but thorough testing must await implementation.

The possibility of *rational errors* makes debugging an important part of any problem solving theory. Rational errors are defined as mistakes in planning that arise from the use of reasonable heuristics. This aspect of our theory is developed by designing *DAPR* (an acronym for *Debugger of Annotated PRograms*), a debugging module for use with PATN. In DAPR terms, *diagnosis* is the isolation of incorrect or incomplete transitions made between ATN states during the planning process. *Repair* consists of re-planning, guided by advice from the diagnosis. A description of basic bug types in terms of specific errors in the planning process is undertaken. DAPR would diagnose and repair *unnotated* programs, in that a record of PATN's planning decisions (the *derivation tree*) is expected to be associated with the code.

Throughout the paper, we employ examples from two benchmark AI domains: the blocks

world and the Logo turtle world [Papert 1971a, 1971b, 1973]. Blocks world problem solvers include SHRDLU [Winograd 1972], BUILD [Fahlman 1974], HACKER [Sussman 1973] and NOAH [Sacerdoti 1975]. Hence, applying PATN to the blocks world provides a common set of problems for comparison. The virtues of the Logo graphics world are: (a) graphics is an environment in which multiple problem descriptions are possible, ranging from Euclidean geometry to Cartesian geometry; (b) the possible programs range over a wide spectrum of complexity; and (c) extensive documentation exists on human performance in this area [G. Goldstein 1973; Okumura 1973].

Section two presents a taxonomy of planning techniques, and uses this taxonomy to construct the planning ATN. This defines a basic problem solver which, due to its reliance on exhaustive backtracking search in traversing the ATN, would be inefficient. The third section addresses this drawback, discussing subtleties in planning when viewed as a search process. Since certain heuristically justifiable planning choices can nonetheless lead to bugs, section four develops a complementary theory of debugging. This allows for reconceptualizing Sussman's [1973] HACKER in section five. The concluding section considers limitations, extensions, and applications of the Structured Planning and Debugging approach.

2. Structured Planning

> For a fortnight I had been attempting to prove that there could not be any function analogous to what I have since called Fuchsian functions. I was at that time very ignorant. Every day I sat down at my table and spent an hour or two trying a great number of combinations, and I arrived at no result. One night I took some black coffee, contrary to my custom, and was unable to sleep. A host of ideas kept surging in my head; I could almost feel them jostling one another, until two of them coalesced, so to speak, to form a stable combination. When morning came, I had established the existence of one class of Fuchsian functions...
> Poincare, H., "Mathematical Discovery," in [Rapport 1963, p. 132]

In this passage, Poincare seems to suggest that human problem solving relies heavily upon laborious consideration of numerous possibilities. Is Poincare correct, or is there a well organized collection of planning concepts to guide the problem solver? Polya's many insightful analyses [1957, 1962, 1965, 1967] support the assertion that planning knowledge is highly structured. This section pursues a view closer to Polya's, by classifying plans, and by developing that classification into a procedural theory of design.

Figure 1 shows an hierarchical taxonomy of common planning techniques.[4] We shall illustrate how an ATN can be used to represent this planning knowledge procedurally, by scrutinizing solution by (a) *identification* with previous solutions and (b) *decomposition* into conjunctive subgoals. The taxonomy shown in the figure is more extensive than this, in order to indicate the context in which our discussion takes place. *Repetition* and *reformulation* are considered, briefly, near the end of this section.

In the taxonomy, planning begins with a choice between three methods -- identification, decomposition and reformulation. By *identification*, we mean recognizing the problem as one which has previously been solved, or noticing that the current problem is a direct special case of one which has previously been solved. By *decomposition*, we mean dividing the current problem into sub-problems which are (hopefully) easier to solve. The third category, *reformulation*, refers to transforming the problem description into an alternative form whose solution is equivalent to, or a

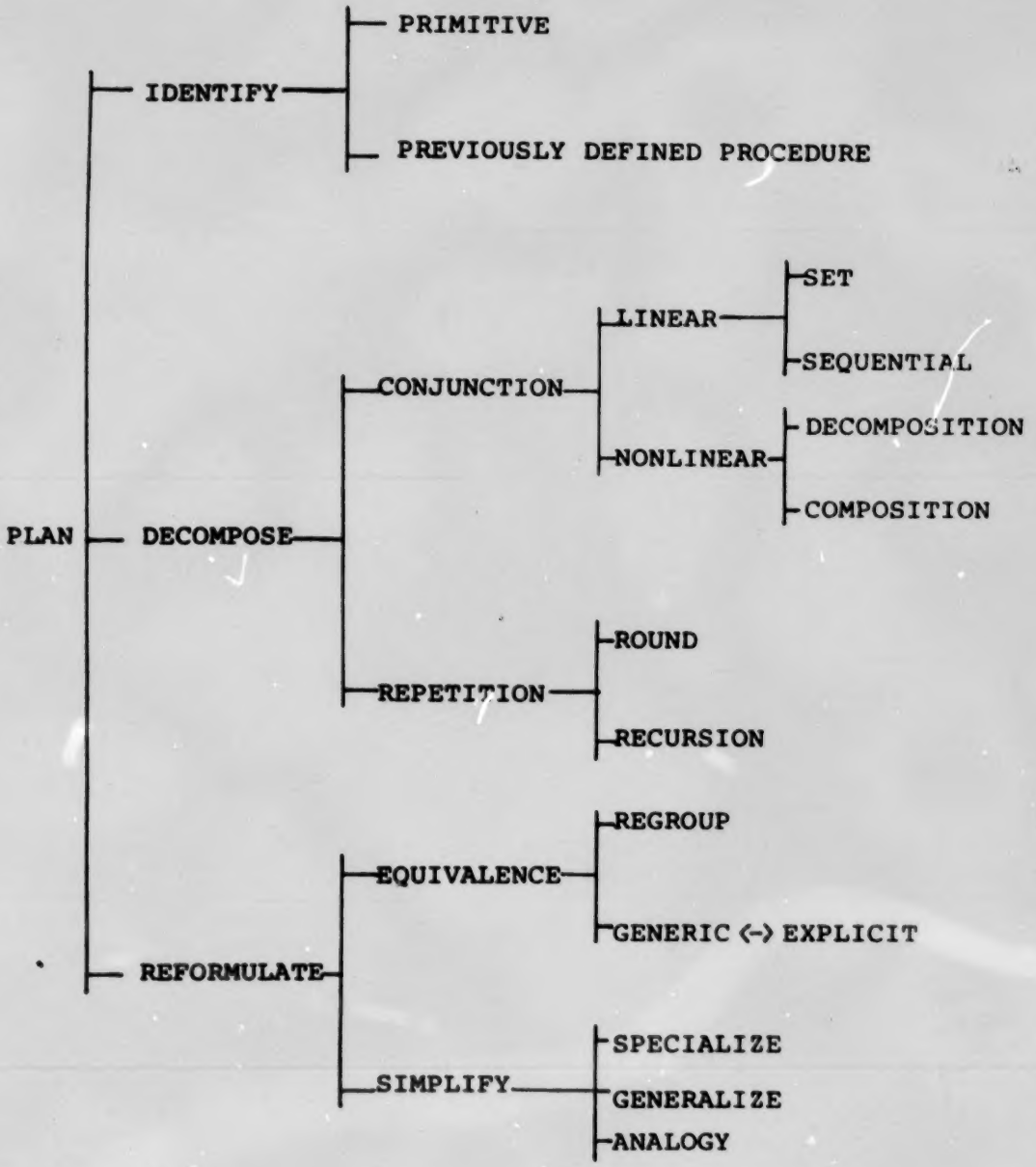


FIGURE 1
TAXONOMY OF PLANNING CONCEPTS

least a stepping stone towards, the solution of the original problem.

2.1. An Augmented Transition Network for Planning

Before presenting additional details concerning our classification of planning techniques, it may be helpful to introduce the manner in which our definitions are formalized via representing them in an ATN. Figure 2 provides a global view of PATN, showing the connections between the various planning states. The stage is set by conceptualizing our planning taxonomy as a *decision tree* of alternative plans. The decision process is modeled by a corresponding *finite state transition diagram*: each named plan type in the taxonomy becomes a *state* in the transition graph; each "subset link" becomes an *arc*.

This planning taxonomy (decision tree) is converted to procedural form by the following augmentations:

(1) *Registers*: Several registers are introduced to carry the semantics of the problem solving process. This includes the specifications for the procedure currently being constructed (Model), and the currently proposed solution (S). Figure 3 is a list of the registers which are used in this report.

(2) *Arc Ordering*: The arcs emanating from each node (representing alternative planning decisions) are ordered, thereby defining a backtracking algorithm. The default ordering from a given node is clockwise, beginning at the entrance point of the incoming arc. This ordering embodies prior judgments about the relative simplicity and probability of success of alternative planning methods.

(3) *Arc Predicates*: The basic arc ordering is supplemented by associating conditions (predicates) with arcs. In the ATN formalism, arc predicates are employed to determine the legality of a transition. By examining the contents of the registers, these arc predicates can make planning choices more sensitive to the problem context.

(4) *Arc Actions*: The contents of the registers may be modified by actions associated with various arcs. The actions are performed if and only if the arc is followed.

(5) *Linearization Cycle*: The nature of nonlinearity is carefully examined, and, as a result, a linearization cycle is introduced. This involves the arcs from the CONJ state to the NLC and NLD states in figure 2. If the arc from CONJ to NLD is followed, for example, the M register is altered to reflect a non-linear

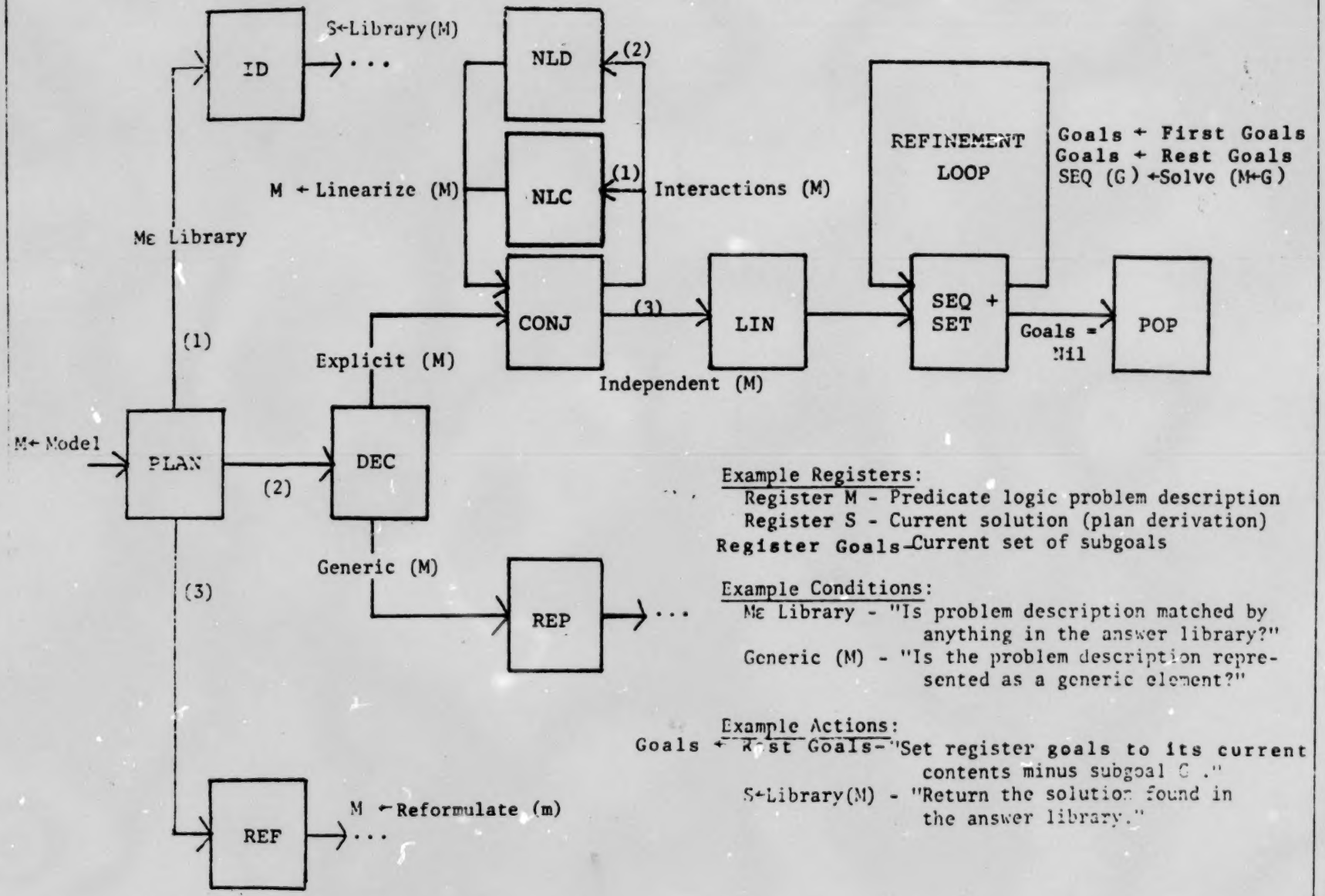


FIGURE 2 - A GLOBAL VIEW OF PATN

Figure 3. Registers Used by PATN

M -- The problem specification or model.

S -- The current solution.

CAVEATS -- A list of warnings regarding possible errors generated during planning to aid in later debugging.

ADVICE -- A list of recommendations to guide the planner in subsequent decisions. Constraints on the order of invoking subprocedures, for example, are recorded in this register.

GOALS -- The set of subgoals whose solutions are currently pending.

G -- The current subgoal, which is about to be solved by a recursive call to PATN.

decomposition (as explained shortly).

(6) *Refinement Loop*: A sequential refinement loop is introduced, which selects a solution order for subgoals and recursively solves for them.

The result of this metamorphosis is the augmented transition network for planning (PATN).

2.2. PATN's Plan Node

> Method consists entirely in properly ordering and arranging the things to which we should pay attention.
>
> Descartes, OEuvres, vol. X, p. 379; "Rules for the Direction of the Mind," (Rule V), from [Polya 1965, p. 77]

PATN's first planning choice involves selection between the major categories of identification, decomposition, and reformulation. We now consider how this first part of the planning process is represented in the ATN (figure 4). The arcs in the figure have been labelled by small letters to facilitate discussion. Arc a begins the planning process by setting M to the formal description of the problem. Arcs b, c and d are the possible transitions from the PLAN state. The default ordering is for IDENTIFICATION to be attempted before DECOMPOSITION or REFORMULATION. This reflects the heuristic judgment that it is preferable to check if the answer is already known before attempting to decompose the problem into subgoals or reformulate the problem description.

Arc b from PLAN to IDENTIFY has an arc constraint. Identification is pursued only if the problem model M can be found in the answer library. If it can, PATN will execute arc e. Here, S, the solution register, is set to the answer found in the library. The POP causes PATN to return with this answer. There are no arc predicates on arcs c or d because DECOMPOSE and REFORMULATE are intended to be applicable to any model. Section 2.3 pursues the discussion by following arc b.

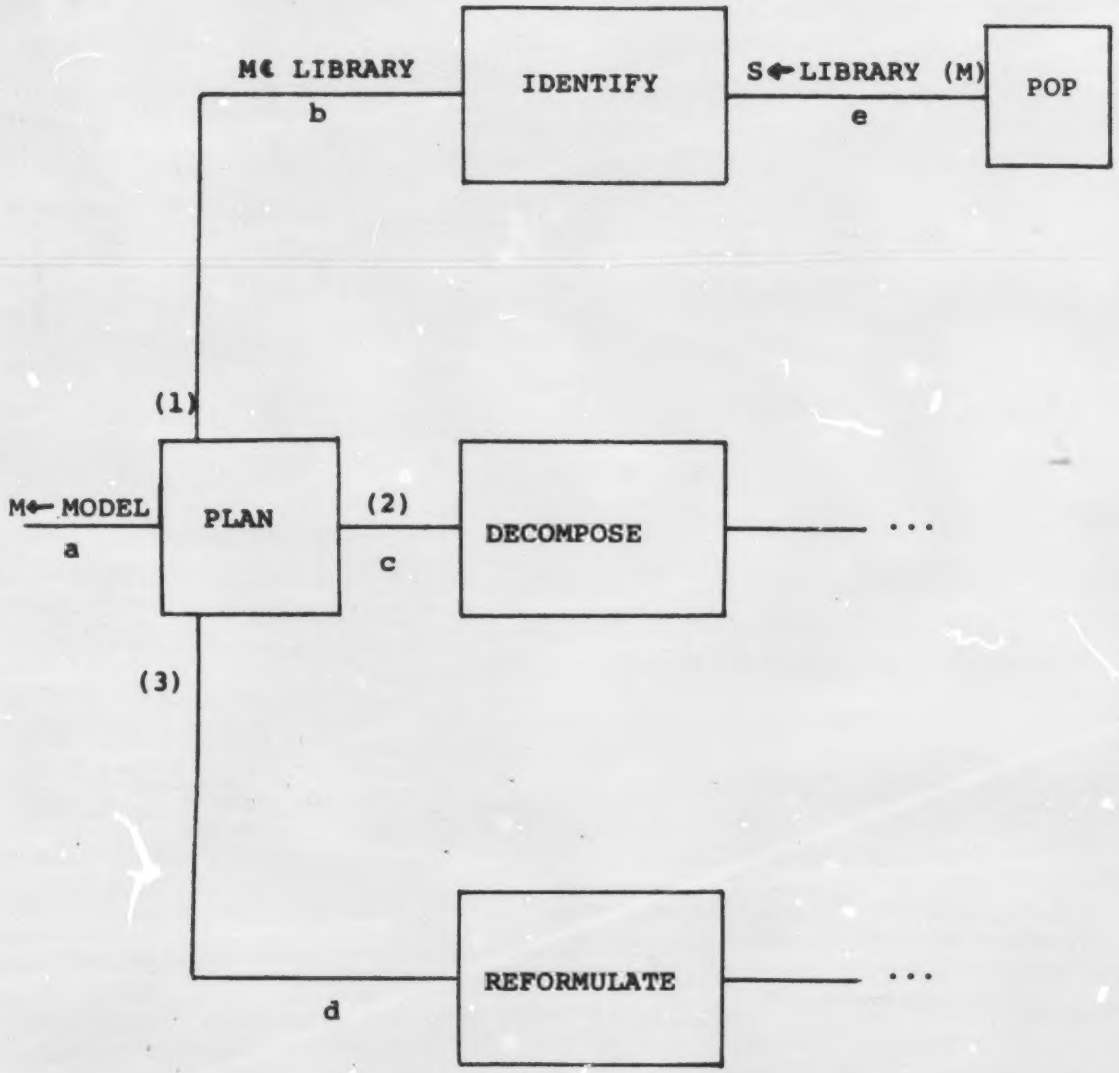


FIGURE 4
PATN's PLAN NODE

2.3. Problem Identification

> From desire ariseth the thought of some means we have seen produce the like of that which we aim at; and from the thought of that, the thought of means to that mean; and so continually, till we come to some beginning within our own power.
> Thomas Hobbes, Leviathon (Chapter III), in [Polya 1965, p. 22].

Problem identification is the minimal technique required for solving problems: retrieval from a library of known solutions. (This is not to say that there are no subtleties involved in designing this component of a procedural problem solver.) The power of the technique arises from: (a) the extensibility of the library; and (b) the manner in which solutions are indexed by their problem descriptions, to facilitate retrieval in appropriate situations.

The answer library is initialized with the primitives provided by the problem domain of interest, described by their effects. Each problem which is subsequently solved is added to the library.[5] The answer library thereby grows in breadth with each successful problem solving episode. As a result, a problem which could not have been solved in reasonable time initially may become realistically solvable later, when one or more of its sub-problems have been solved and added to the library.

To develop problem identification rigorously, a precise description of primitives and problems is required. For our purposes in this report, a problem is represented as *a conjunction of assertions about a set of objects, their properties, and their relationships.* This formal problem specification is called the *model.*[6] This is a traditional method: although we use a different notation which we find more convenient, our models could be straightforwardly translated into statements in the predicate calculus.[7]

Entries in the answer library have two parts: a *Pre Model*, and a *Post Model*. The *Pre* Model is a conjunction of assertions which are prerequisites for the entry: the entry is not guaranteed to work if these assertions are not satisfied. The *Post* Model is a conjunction of assertions which describe the *effect* of the entry: the goal which it is to accomplish. (This

approach is analogous to the definition of *operators* in STRIPS [Fikes et al. 1972].) For PATN, primitives and problems are *indexed* by their Post models only.[8]

To illustrate the use of predicate models for indexing the answer library, let us consider the primitives for the Logo turtle world. The turtle is a graphics cursor on a display that is moved primarily by two commands: FORWARD and RIGHT. The former moves the turtle display in the direction of its current heading. The latter rotates the turtle around its own axis. Like any problem, the primitive FORWARD is described in the answer library by a *Post* Model that indicates its effects, i.e., what it can be used for, and a *Pre* model that states its prerequisites.

Pre Model for (FORWARD X)

To execute (FORWARD X), two objects must exist: a turtle and a display. These two objects must satisfy the relationship that the new position for the turtle (as specified by the Post Model) lie within the boundaries of the display.

```
(EXISTS TURTLE)
(EXISTS DISPLAY)
(< |(XCOR (FORWARD X))| Xmax)
(< |(YCOR (FORWARD X))| Ymax)
```

Post Model for (FORWARD X)

The result of executing (FORWARD X) is that there exists a vector with length X, whose direction is the previous heading of the turtle and whose visibility is the previous state of the pen. (Dots (".") are used to indicate the previous value.)

```
(EXISTS VECTOR V)
(= (LENGTH V) X)
(= (XCOR TURTLE)
   (+ :(XCOR TURTLE) (* X (COS :(HEADING TURTLE)))))
(= (YCOR TURTLE)
   (+ :(YCOR TURTLE) (* X (SIN :(HEADING TURTLE)))))
(= (HEADING V) :(HEADING TURTLE))
(= (VISIBILITY V) :(VISIBILITY TURTLE))
```

Problems are represented similarly. Figure 5, a "wishingwell picture," is a typical scene that a Logo student might attempt to accomplish by manipulating the turtle. This kind of project is commonly undertaken by beginners after two to five hours of experience with the computer

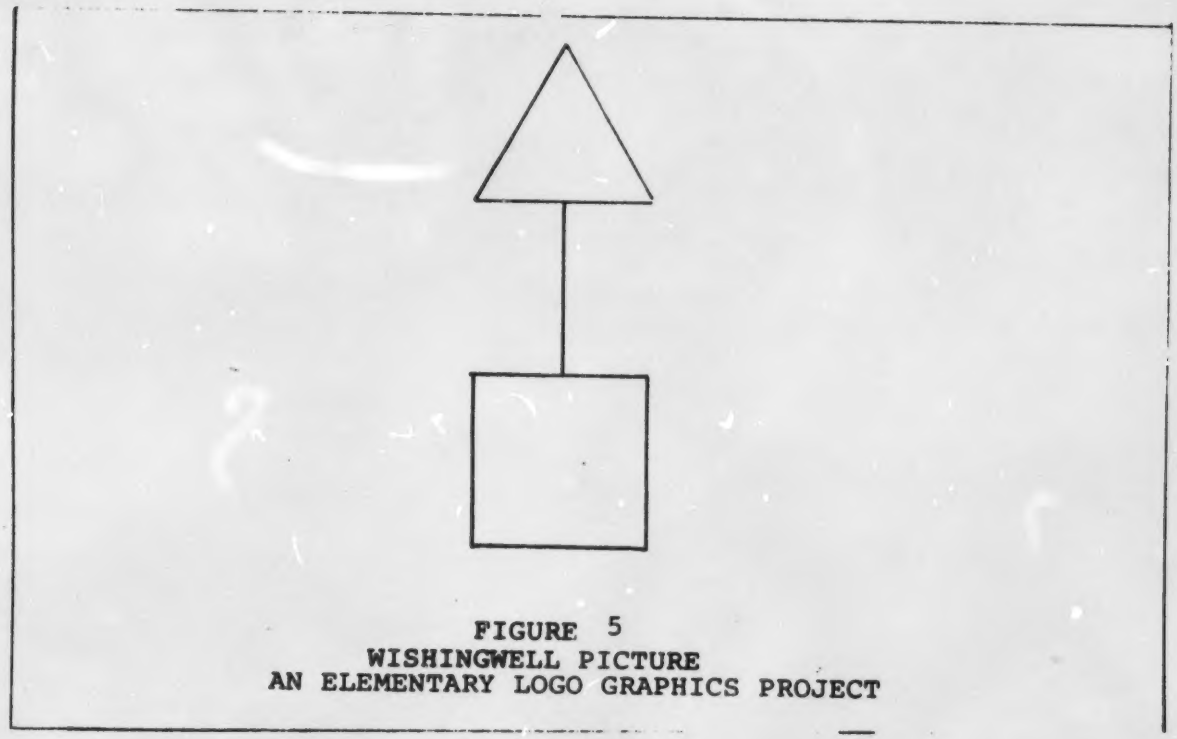


FIGURE 5
WISHINGWELL PICTURE
AN ELEMENTARY LOGO GRAPHICS PROJECT

[G. Goldstein 1973, p. 23]. An English statement of the problem might be: *draw a wishingwell with a square well and a triangular roof.* To allow formal treatment of this problem, we use a predicate (Post) model of the desired picture as input to PATN. The model is expressed in a simple assertional formalism developed by Goldstein [1974].[9]

```
MODEL WISHINGWELL
1 PARTS ROOF POLE WELL
2 TRIANGLE ROOF;   3 LINE POLE; 4 SQUARE WELL
5 ABOVE ROOF POLE; 6 ABOVE POLE WELL
7 CONNECTED WELL POLE (AT P)
   8 (= P (MIDDLE (UPPER (SIDE WELL))))
   9 (= P (LOWER (ENDPOINT POLE)))
10 CONNECTED POLE ROOF (AT Q)
   11 (= Q (MIDDLE (BOTTOM (SIDE ROOF))))
   12 (= Q (UPPER (ENDPOINT POLE)))
13 HORIZONTAL (BOTTOM (SIDE ROOF))
14 HORIZONTAL (UPPER (SIDE WELL))
END
```

Later we attempt to show that the particular choice of model is not critical: PATN has been designed to utilize a variety of heuristics for reformulating the model if necessary.[10]

For the blocks world, the basic instruction to the one-armed robot is (PUTON A B), where A and B are blocks. The Pre Model is:

To execute (PUTON A B), A must have a clear top, in order to be picked up. A must be at some known old position. Also, the top of B must have enough room for A.

```
(CLEARTOP A)
(ON A OLD-POSITION)
(SPACE-FOR A B)
```

The Post Model asserts:

Block A is no longer on its old position. A is on B. Also, the top of B is not clear.

```
(NOT (ON A OLD-POSITION))
(ON A B)
(NOT (CLEARTOP B))
```

For basic blocks world problems, the model is simply a conjunction of ON relationships. For example, a tower of three blocks would have the model:

(AND (ON A B) (ON B C)).

2.4. Problem Decomposition

> Divide each problem that you examine into as many parts as you can and as you need to solve them more easily.
> Descartes, OEuvres (vol. VI), p. 18; "Discours de la Methode" (Part II).
>
> This rule of Descartes is of little use as long as the art of dividing ... remains unexplained .... By dividing his problem into unsuitable parts, the unexperienced problem-solver may increase his difficulty.
> Leibnitz, "Philosphische Schriften," edited by Gerhardt, vol. IV, p. 331.
>
> From [Polya 1965, p. 129].

Our theory of planning addresses Leibnitz' criticism of Descartes by developing more precisely the nature of decomposition techniques. The planning taxonomy identifies two important methods: *conjunction* and *repetition*. The first type of plan is appropriate for achieving a goal which is described as a simple conjunction of predicates (such as the three-high Tower above). The second plan type is appropriate for achieving a goal which is described as a particular subgoal repeated some number of times (such as a Tower of arbitrary height). In conventional programming languages these plan types are implemented using sequencing and iteration (or recursion), although other language constructs are possible (such as parallelism).

PATN's decision to pursue CONJUNCTION versus REPETITION is based on the form of the model. For our purposes here, a given sub-model is restricted to being either *explicit* or *generic*. The former has an explicit list of parts. Wishingwell is an example of such a model. The latter uses quantification to describe the overall model in terms of a "generic" part. EQUITRI1 and EQUITRI2 given below are two equivalent models for an equilateral triangle. The first is explicit while the second is generic.

```
MODEL EQUITRI1
1 PARTS S1 S2 S3 R1 R2 R3
2 LINE S1; LINE S2; LINE S3
3 ANGLE R1, ANGLE R2, ANGLE R3
4 S1 = S2 = S3
5 R1 = R2 = R3
6 CONNECTED S1 S2
7 CONNECTED S2 S3
8 CONNECTED S3 S1
END
```

```
MODEL EQUITRI2
1 PARTS (S 3) (R 3)
2 FOR-EACH I, LINE S(I)
3 FOR-EACH I, ANGLE R(I)
4 FOR-EACH I, I=1,3,
    S(I) = S(I+1 MOD 3)
5 FOR-EACH I, R(I) = 120
6 FOR-EACH I,
    CONNECTED S(I) S(I+1 MOD 3)
END
```

Because of the simplifications we have introduced, generic models can be trivially distinguished from explicit models by the presence of the quantifier "FOR-EACH." In the general case, models could be arbitrary logical expressions with mixed existential and universal quantification. The elementary blocks world tasks and Logo picture problems which we are considering do not require this complexity. (A direction for future research is to extend PATN's design to handle these more complex problem descriptions.)

The ATN representation for this decision is illustrated by figure 6. Examine the DECOMPOSE node. The transition to CONJUNCTION is made only if the problem is described by a model with explicit parts such as EQUITRI1 or WISHINGWELL. If the model is constructed from a generic description as in EQUITRI2, then REPETITION is selected. Thus, in terms of arc predicates, the alternatives at the DECOMPOSE node are mutually exclusive. It is possible that a REPETITION plan, for example, might eventually be produced for a problem initially described by an explicit model. However, this would occur only through an intermediate REFORMULATION in which EQUIVALENCE converted the original model to generic form. This in turn would allow a recursive call to PATN in which DECOMPOSITION would then choose REPETITION.

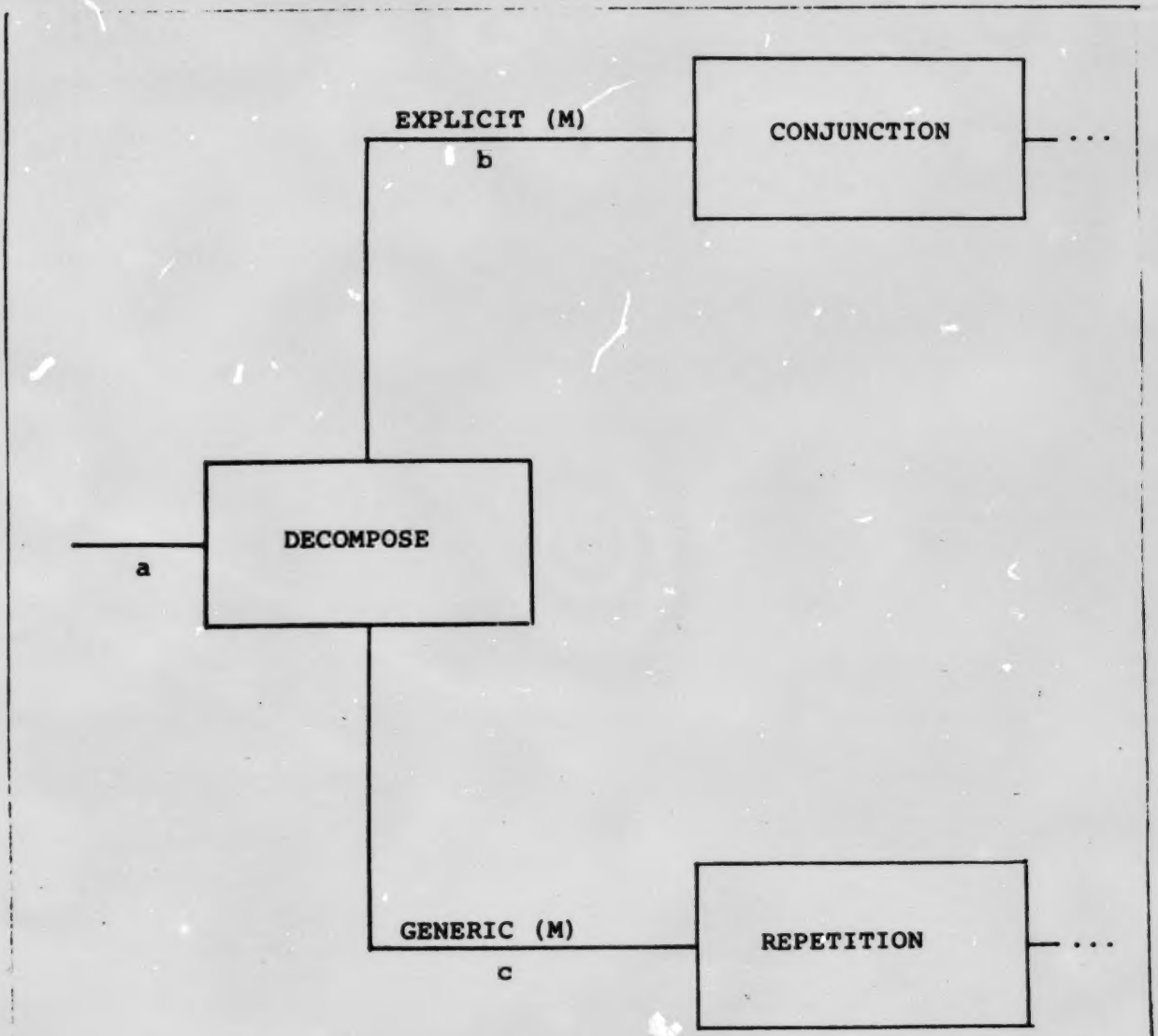


FIGURE 6
PATN's DECOMPOSE NODE

2.5. Decomposition by Conjunction

For conjunctive plans, PATN's next design decision is whether the conjuncts are to be treated independently, or, alternatively, whether notice must be taken of interactions. For example, a *linear* plan for the wishingwell of figure 5 would solve for the three sub-pictures -- ROOF, POLE and WELL -- as separate subprocedures, each constructed independently of the others. A *nonlinear* plan might attempt to take account of the potential interactions between POLE and the other parts -- modifying the specifications for ROOF and WELL so as to start and stop in the middle of a side -- facilitating connection with the POLE. However, since such an optimization requires that a given subprocedure incorporate knowledge regarding the implementation of another subprocedure, a *linear* plan would not do this.

Let us be more precise in our classification of nonlinearities. The goal is to construct a procedure to accomplish a conjunction of assertions. *Nonlinearities in decomposition* are those that add constraints to the design of the subprocedures. *Nonlinearities in composition* (i.e., in putting the parts back together) are those that add constraints to the design of the superprocedure.

For the wishingwell example, adding the constraint to the design of the subprocedures for the ROOF and WELL that they start in the middle of a side is an example of a nonlinear decomposition. Another example for the Logo world -- which involves more than optimization -- occurs for problems which specify that one object, X, is to be INSIDE another, Y. Y must be larger than X, if the required topological relation is to hold. This means that a linear decomposition that ignores the INSIDE relation and draws Y to some default size is likely to fail. The correct approach is to add a SIZE property to the descriptions of both X and Y.

A *nonlinear composition* adds constraints to the design of the superprocedure. For the blocks world, the most common form of this nonlinearity is the existence of a *partial ordering on the sequence in which the subgoals should be achieved.* The ordering constraints arise from the use of some temporary resource (such as space), by one subgoal, which is eventually used in a conflicting

way by another. An example discussed by Sussman [1973] and Sacerdoti [1975] is the construction of a tower of three blocks, i.e., (AND (ON A B) (ON B C)). The tower must be built from the bottom up if the subgoals are not to conflict. The constraint (BEFORE (ON B C) (ON A B)) must be added to the design of the superprocedure.

The same kind of nonlinearity can arise in ~ Logo animation. To create a "snapshot" of some picture which can be displayed anywhere on the screen, the picture must first be drawn and "photographed." This process, called "snapping," involves first drawing the picture, next snapping it, and then erasing it. Now the erasure is of an entire screen region. If another shape is present it will be destroyed. Hence, no shapes intended to appear in the final scene should be present. Thus, a constraint must be asserted that requires that the snapping subgoal be achieved before any subgoals that draw a permanent shape in the critical screen region.

Nonlinear decomposition and composition constraints are not mutually exclusive. A given problem can exhibit both kinds of interactions. In the next sub-section, we take account of this by including a cycle in the ATN that progressively linearizes each interaction detected in the model.

2.6. PATN's Subgraph for Conjunction

Figure 7 shows PATN's subgraph for conjunction. Arc b from CONJUNCTION to LINEAR decomposes the model into sub-models that will be solved for independently by recursive calls to PATN. This is done as follows. Two classes of sub-models are created. One class describes the individual objects. The second class describes interactions between pairs of objects.

- For each object Xi in the model M, Mi is the sub-model composed of all the assertions in M describing properties of Xi.
- For each pair of objects Xi and Xj in the model M, Mi,j is the sub-model composed of all the assertions describing relations between Xi and Xj.

We speak of accomplishing the object Xi described by Mi as a *main step* in the overall procedure. Relations between two objects described by Mij are accomplished by *interface steps*.

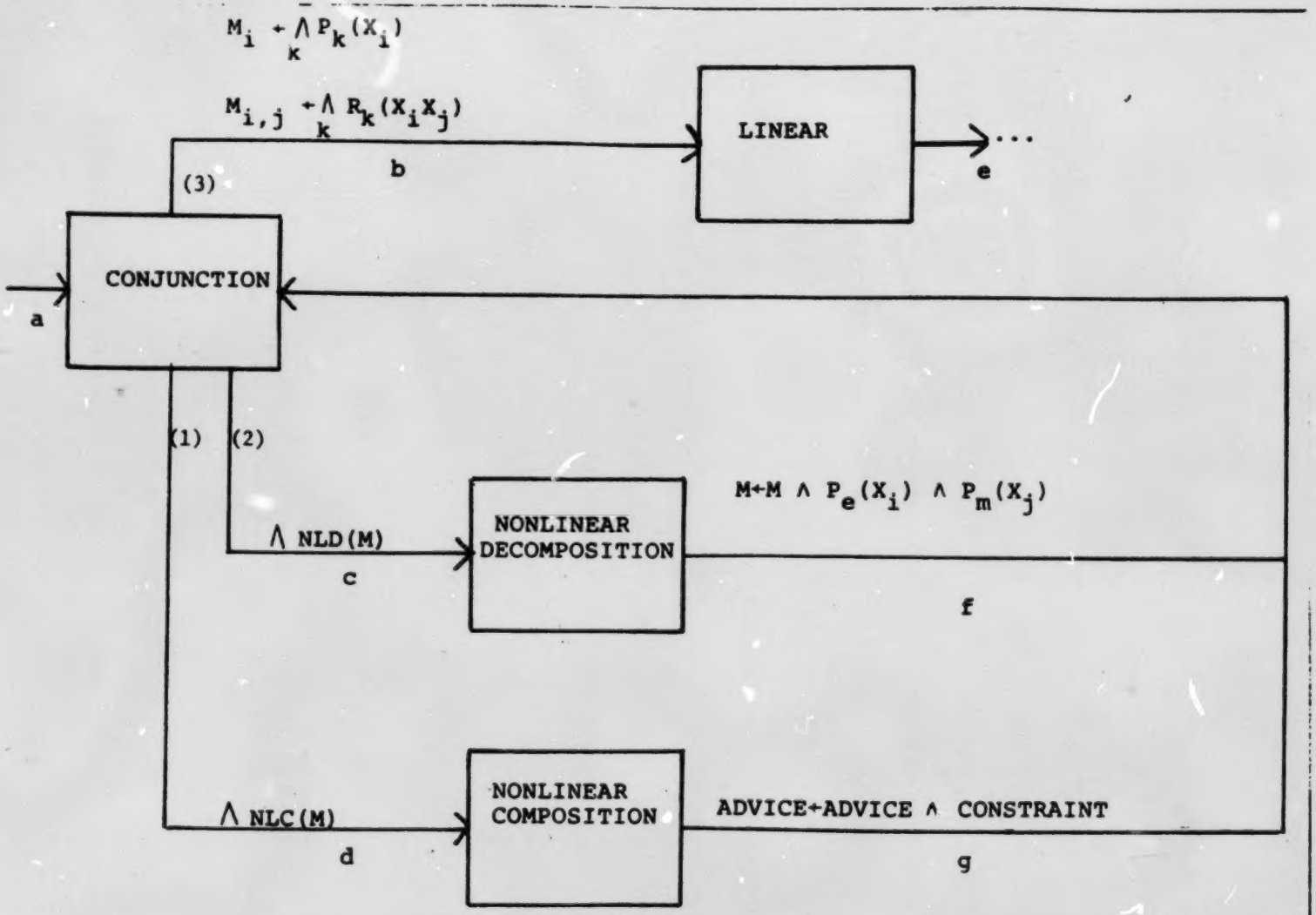


FIGURE 7

PATN's CONJUNCTION NODE

As an example, a linear decomposition of the wishingwell is:

```
M1: TRIANGLE ROOF;
    HORIZONTAL (BOTTOM (SIDE ROOF))

M2: LINE POLE;

M3: SQUARE WELL;
    HORIZONTAL (UPPER (SIDE WELL))

M1,2: ABOVE ROOF POLE;
      CONNECTED POLE ROOF (AT Q)
      (= Q (MIDDLE (BOTTOM (SIDE ROOF))))
      (= Q (UPPER (ENDPOINT POLE)))

M2,3: ABOVE POLE WELL
      CONNECTED WELL POLE (AT P)
      (= P (MIDDLE (UPPER (SIDE WELL))))
      (= P (LOWER (ENDPOINT POLE)))
```

We define *linear sequential refinement* as solution by the following process:

(1) organize the mainsteps into a sequential procedure, choosing an ordering that satisfies any *linearization advice*;

(2) solve for the mainsteps independently;

(3) solve for the interfaces in the order in which they occur in the procedure.

A linear decomposition is valid if a corresponding solution via linear sequential refinement is possible. Implicit interactions can invalidate a linear decomposition.

The *linearization cycle* consists of arcs c,f and arcs d,g. These arcs attempt to *linearize* the model by checking for known types of interactions. The nonlinear decomposition node adds properties to the descriptions of individual subgoals that take account of interactions. The nonlinear composition node sets an advice register that will be accessed by the SEQ operator (explained below) in constructing the superprocedure.

NLD is a conjunction of conditions (predicates), each of which checks the model for a particular relation or pattern of relations that have nonlinear consequences for the decomposition.

If any of these predicates detect their kind of interaction, properties are added to the description of individual objects that explicitly account for the dependency. The objective is that with these additional properties an independent treatment of the modified object specifications will be successful.

For example, as discussed above, INSIDE is a relation in the turtle world that has consequences for the properties of the objects involved. Thus, NLD-INSIDE checks for the existence of (INSIDE X Y) in the model. If found, SIZE properties, describing X in terms of Y, or Y in terms of X, or both, are added to the properties of these objects. The result is that an independent solution for (the revised versions of) X and Y will not prevent the INSIDE relation from being accomplished.

NLC checks for patterns in the model that have consequences for the eventual composition of the subgoals. If such properties are detected, then explicit relations are added to take account of the interactions. An example is NLC-ANIMATION that checks for a Logo animation that creates snapshots and shows them. If detected, (BEFORE SNAP DISPLAY) is appended to the contents of the ADVICE register. Similarly, for the blocks world, NLC-TOWER adds (BEFORE (ON B C) (ON A B)) to ADVICE.

The NLC and NLD constraints arise from two sources. The first is that they may be supplied by the creator of the Planning ATN. Alternatively, following Sussman [1973], PATN can be designed to summarize bugs by classifying the nature of the nonlinearity and adding it to the NLC and NLD constraints. In these terms, the acquisition of skill is, at least partly, the growth of more elaborate recognition routines for implicit interactions. Sussman called this process the compilation of critics. The theoretical advance of Structured Planning over Sussman's HACKER paradigm is to make clear that these critics are simply additional arc constraints in the planning transition graph. They are not different in kind from any other planning constraints.

To summarize, implicit dependencies are handled by the ATN's linearization loop. If the

problem is identified as involving some kind of nonlinearity, then the model or advice registers are modified to make the interaction explicit. Processing then returns to the CONJUNCTION node. Further processing of interactions occurs, until no more are detected. Control then passes to the LINEAR node for actual decomposition. If an interaction still exists, but has gone undetected, subsequent debugging will be necessary.

2.7. Composition by Sequential Refinement

Once the nonlinearity loop has been completed, PATN would go on to solve the individual subgoals and compose a complete solution. In this section, we discuss a composition technique we term *sequential refinement*. A generalization of this approach, *net refinement*, based on the procedural net representation for programs, is discussed in section three.

Figure 8 illustrates the ATN subgraph for the sequential refinement cycle. The basic process is cycling through the subgoals identified by the linear decomposition and solving for each by recursive application of the ATN. Arc b enters the sequential refinement loop. The solution register S is set to a sequential superprocedure for the mainsteps Mi and interface steps Mij identified by the decomposition. The SEQ operator on the arc chooses an order for this superprocedure that is consistent with any ADVICE recorded by the linearization loop. SEQ might also bring additional criteria to bear on the organization of the superprocedure, such as imposing an order that mirrors chains of predicates in the model, such as X connected to Y connected to Z. This often simplifies interfacing.

As an example, for the wishingwell problem, given the Mi and Mij specified above, a plausible sequence of mainsteps would be:

```
TO WW
10 ROOF
20 POLE
30 WELL
END
```

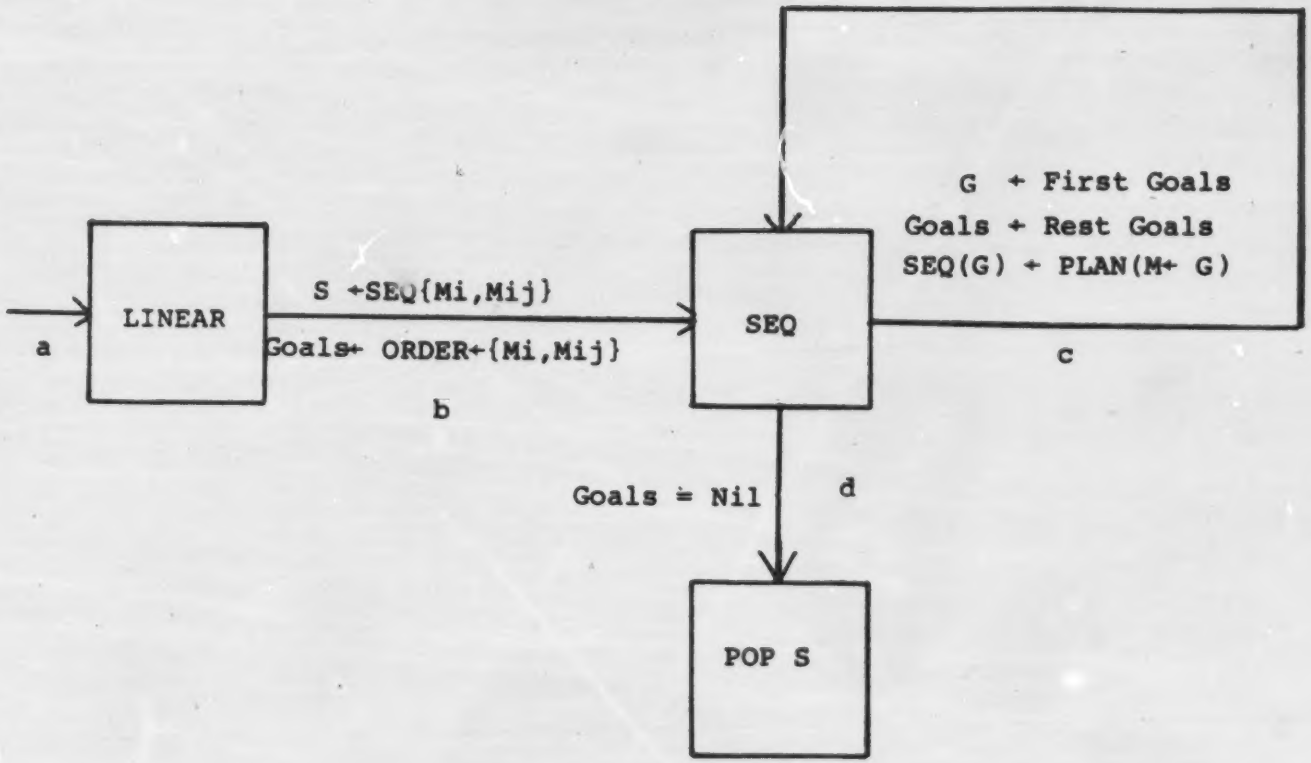


FIGURE 8
SEQUENTIAL REFINEMENT

The ORDER operator on arc b of figure 8 chooses the sequence in which the sub-problems are solved. This may not be, indeed, probably is not, identical to the order of occurrence of the sub-problems in S. A criterion for the order of solution, for example, is to solve for the mainsteps before the interfaces. Another criterion is to order the mainsteps with respect to their complexity. *LookAhead* (section three) can estimate this. For the wishingwell, it makes sense to solve for the POLE first since lookahead can identify this as a primitive. Criteria for ordering the relations can exist as well, although the default ordering is usually the order of occurrence in the procedure.

Arc c is a cycle that recursively solves for the subgoals in the order selected by ORDER. The solution for each subgoal is attached to S at that subgoal's node. The solved subgoal is then deleted from GOALS. When all subgoals have been solved, the cycle is exited via arc d. The ATN pops, returning the solution.

For increased effectiveness, PATN's initial Logo world answer library would contain both primitives with their associated models as well as schemata for accomplishing particular model relations. Thus, if the sub-problem is to achieve (ABOVE X Y), where X and Y are mainsteps that have already been solved, then the answer library would contain specific procedural knowledge for designing an interface, relative to the adjacent mainsteps, that satisfies that relationship. The nature of these imperative schemata is discussed in [Goldstein 1974, Appendix D]. We do not give details here.

For the wishingwell, the mainsteps for the ROOF, POLE and WELL would be solved first. Then, pursuing the default order for relations, first the interface between ROOF and POLE and then between POLE and WELL would be constructed. Figure 9 shows PATN's solution (as hand-simulated by the authors) and the sequence of ATN states involved in its generation.

Besides generating the program, PATN would generate its annotation, an hierarchical trace of the ATN states passed through in generation. In this *derivation tree*, each node has a copy of the values assigned to the registers at the time the node was generated. This serves as a description of

PLAN(M ← WW)

→DECOMPOSE

→CONJUNCTION

→LINEAR

→SEQ*

→POP(S)

WHERE S IS:

TO WW

10 M_1 ← TRI ← ROOF ←

20 $M_{1,2}$ ← BELOW, CONNECTED ←

30 M_2 ← LINE ← POLE ← WW

40 $M_{2,3}$ ← BELOW, CONNECTED ←

50 M_3 ← SQ ← WELL ←

FIGURE 9

SOLVING THE WISHINGWELL PROBLEM

the purposes of the code in the form of the MODEL assignments, ADVICE for future modifications and CAVEATS regarding possible bugs. Caveats are generated by the planner when making possibly erroneous heuristic decisions; these are discussed in a later section on debugging. The derivation tree for the wishingwell procedure (abbreviated slightly) is illustrated in figure 10.

2.8. Decomposition by Repetition

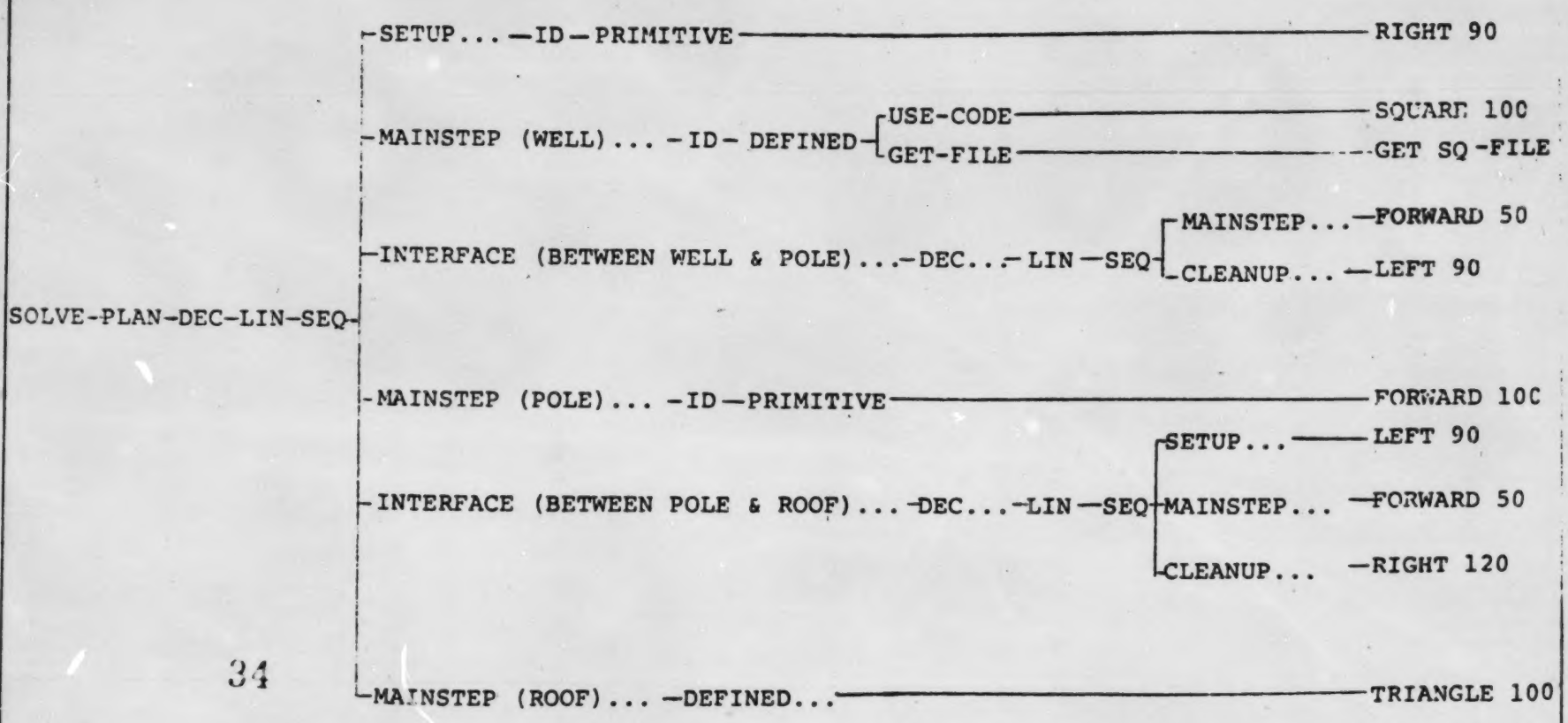
Before concluding this section, we briefly consider other planning techniques which were illustrated in our taxonomy but have not been elaborated in the discussion so far. *Repetition plans* correspond to the problem solving method of structuring the solution in terms of either the same goal applied to simpler arguments (recursive plans) or another simpler goal repeated some number of times (round plans). The former technique is more powerful than the latter in the sense that every round plan can be accomplished by means of a recursion, while every recursion cannot be accomplished by iteration [Hewitt 1972]. But Round plans are differentiated because the problem formulation which would trigger them for PATN differs from that of Recursive plans. In the former case, the problem P is described as n repetitions of problem Q, where Q≠P, while in the latter P is described in terms of repeated occurrences of problem Q=P.

Round plans are the natural planning technique for generic models. We intend to handle this in the ATN via an arc operator ROUND that formulates a sub-model for the generic part and advice for the composition requesting an iterative control structure. Having decomposed the problem in this fashion, control would then pass to the Sequential Refinement Loop. Figure 11 illustrates this subgraph.

EQUITRI2 was an example of a generic model. The ROUND operator would isolate subgoals for accomplishing a SIDE and a ROTATION. The repetition advice would be for three iterations. The result would be a program of the following form:

FIGURE 10

ABBREVIATED HIERARCHICAL DERIVATION TREE FOR WISHINGWELL


SOLVE-PLAN-DEC-LIN-SEQ
SETUP... —ID— PRIMITIVE — RIGHT 90
MAINSTEP (WELL)... —ID— DEFINED — USE-CODE — SQUARE 100
GET-FILE — GET SQ-FILE
INTERFACE (BETWEEN WELL & POLE)... —DEC... —LIN —SEQ — MAINSTEP... —FORWARD 50
CLEANUP... —LEFT 90
MAINSTEP (POLE)... —ID —PRIMITIVE — FORWARD 100
INTERFACE (BETWEEN POLE & ROOF)... —DEC... —LIN —SEQ — SETUP... — LEFT 90
MAINSTEP... —FORWARD 50
CLEANUP... —RIGHT 120
MAINSTEP (ROOF)... —DEFINED... — TRIANGLE 100

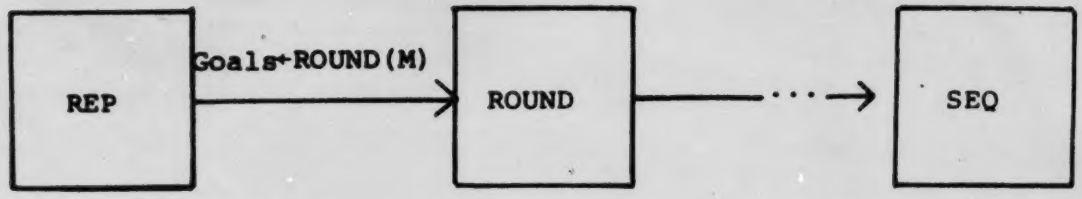


FIGURE 11
ROUND PLANS

```
TO TRI
10 REPEAT 3 20, 30
20 FORWARD 100
30 RIGHT 120
END
```

2.9. Problem Reformulation

> When a problem arises, we should be able to see soon whether it will be profitable to examine some other problems first, and which others, and in which order.
>
> Descartes, OEuvres (vol. X), p. 381; "Rules for the Direction of the Mind" (Rule VI), from [Polya 1965, p. 36].

The final category of plans which we consider consists of techniques for problem reformulation.[11] The importance of these methods can be understood by recognizing that all of the problem solving strategies mentioned above are triggered by pattern matching against the description of the problem. The reformulation techniques, however, are designed to alter the problem description. PATN would apply these reformulation techniques should solution by identification or decomposition fail. Their action is to reformulate the problem description, and then to pass the new description back to the Planner.

Our taxonomy includes two reformulation techniques.[12] The first attempts to find an *equivalent* problem that will be easier to solve, and whose solution will satisfy the original task. The second searches for a *simplification* that can be used as a stepping stone to solving the original problem.

The difficulty in applying reformulation plans lies in recognizing which reformulation will aid the solution progress. For *equivalency*, we envision PATN as capable of reformulations that move between descriptions given in terms of multiple objects to equivalent descriptions in terms of a single generic object, thus changing from a Conjunctive decomposition to a Repetitive decomposition, or vice versa. An example is moving between the EQUITRI1 and EQUITRI2 triangle models. Another reformulation technique involves regrouping the parts. Figure 12 shows

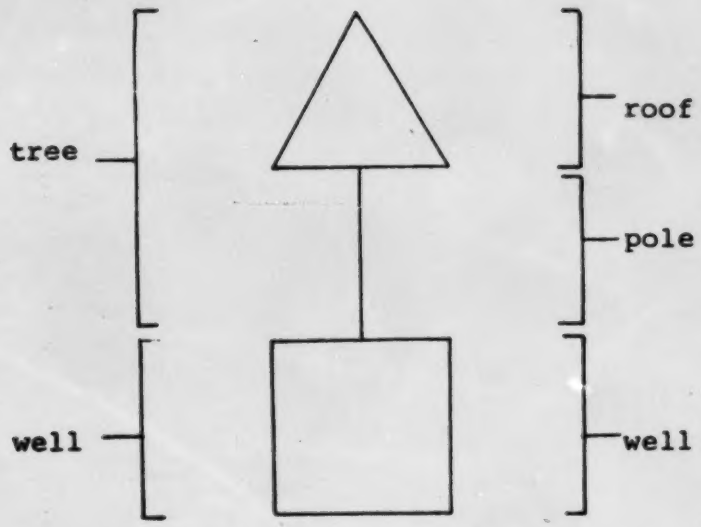


FIGURE 12
REFORMULATING THE WISHINGWELL IN TERMS OF A TREE

an example of parts regrouping for the wishingwell. The virtue of regrouping is that it might produce a model whose parts are already in the answer library.

For Simplification plans, we have analyzed elementary techniques based on generalization, specialization, and analogy. *(a)* Specialization typically involves instantiating variables in a model by specific constants or restricting their range. *(b)* Generalization would include the opposite processes. Other non-equivalent reformulations involve adding or deleting model predicates. *(c)* Analogy often amounts to first generalizing and then specializing. Thus, for the Logo world, if the original model were for a triangle with sides of a certain size, generalization might produce a model for a polygon, or for a triangle of arbitrary size. Analogy might then respecialize to a square, perhaps, or a triangle of another size. The virtue of these reformulation techniques is the possibility for reaching a problem description whose solution is known. We envision that each technique would have associated with it an inverse mapping on the solution so that it can be mapped back to suggest a plan for solving the original problem.

3. Searching for the Plan

> If you see several plans, none of them too sure, if there are several roads diverging from the point where you are, explore a bit of each road before you venture too far along any one -- any one could lead you to a dead end.
> Polya, Mathematical Discovery, (vol. II), p. 27.

The most straightforward plan generation algorithm for PATN is to attempt arc transitions in *depth first* order, with alternatives stored on a backup list. If some plan leads to a subgoal that cannot be solved, failure occurs. Control backs up to the more recent choice point, and planning resumes by pursuing the next untried alternative for that choice point (provided that it is allowed by any arc transition constraints which may be present).

This depth first search would apply to both explicit and implicit choice points in planning. An example of an *explicit* choice is the decision between decomposition and reformulation for a given problem. By *implicit choice point*, we mean those decisions which arise which are not represented as mutually exclusive arc transitions in the ATN. Implicit choices occur in identifying past solutions (more than one previously solved problem may match the Post model for the current problem); creating super-procedures (there may be more than one reasonable sequence); and, in general, whenever knowledge *on* the transition arcs sets registers and makes decisions. We have discussed arc ordering and predicates associated with the transition arcs to direct explicit choices in the planning ATN. For each implicit decision, a similar approach is possible. The decision process locally determines the order of the alternatives, pursues the first, and pushes the remainder onto a failure stack. Thus the overall planning process would remain a depth first search.

Ultimately, all plans which PATN is capable of generating would be tried in this mode. Of course, exhaustive backtracking search is not a practical planning technique. One way of decreasing aimless search which has been discussed is to provide additional constraints on the transition arcs. This section outlines further techniques germane to resolving planning decisions.

These techniques operate by superimposing an executive search process on the ATN so as to improve the efficiency of plan generation. The techniques represent four milestones in the development of a successful planning theory. These improvements, designed to make the planning process more directed and less susceptible to blind search, are: (a) *lookahead* (e.g., [Aho & Ullman 1972]); (b) *least commitment* (e.g., [Sacerdoti 1975]); (c) *differential diagnosis* (e.g., [Rubin 1975]); and (d) *lemma libraries* (cf., *macrops* [Fikes et al. 1972], *well-formed substring tables* [Kuno 1967], [Woods et al. 1972]). We intend to incorporate these strategies into the basic PATN problem solver, following its initial implementation.

3.1. Lookahead

Lookahead consists of a limited search of available alternatives, with associated static plausibility criteria for judging the probable success of a given non-terminal state. An elementary but useful form of lookahead could be accomplished in PATN by pushing the planning process forward some fixed number of recursive levels, looking to see if a solution arises via identification. Thus, a decomposition that can solve most of its subgoals in terms of the answer library would be preferred to a decomposition that must recursively apply decomposition techniques to its subgoals. In effect, such lookahead attempts to select those plans that accomplish the goal with a minimum number of recursive calls to the problem solver.

For example, reconsider the wishingwell scenario. Suppose the answer library contains, not a TRIANGLE program, but a TREE procedure. Lookahead could prevent the planner from blindly pursuing a decomposition in terms of ROOF, POLE, and WELL, over a reformulation that describes the wishingwell as a TREE and a WELL (figure 12). This would be accomplished by observing that the reformulation produces a problem description whose decomposition can be partly solved by means of the answer library; whereas the standard decomposition results in two subgoals (the ROOF and the WELL) that require further analysis.

Lookahead could be implemented in the usual fashion (see, e.g., [Aho & Ullman 1972]). A static plausibility function might assign a plausibility of *one* to problems that can be solved via identification, and *zero* to problems that require decomposition or reformulation. Lookahead would push the analysis through a fixed number of levels of recursion, and then estimate the dynamic plausibility as the sum of the static plausibilities of the subgoals appearing as the tips of the problem tree, divided by the number of these subgoals. The division serves the purpose of resolving the following situation: given two situations in which the same number of subgoals are known, the problem tree with fewer unsolved subgoals is to be preferred.

A refinement of this plausibility computation might assign greater weight to those plans that led to identifications for more complex subgoals. The complexity of a subgoal could be approximated by syntactic criteria such as counting the number of predicates involved.

3.2. Least Commitment

Least commitment is the problem solving technique of avoiding premature decisions. It is elegantly developed by Sacerdoti [1975] in the form of *procedural nets*. Sacerdoti observes that some bugs in planning can arise from premature commitment to a particular sequence, when the available evidence does not in fact require such a determination. His solution is to represent the program, not in the usual sequential format, but as a net.[13]

Figure 13 illustrates a procedural net for building a tower from three blocks. Sacerdoti's planning system, NOAH, proceeds by successively expanding subgoals, committing the system to a sequence only when a conflict in ordering arises. At levels 1, 2 and 3, no order has been chosen for the sequence of accomplishing (ON A B) and (ON B C). It is not until *after criticism at level 3* that NOAH commits itself to an order for placing the blocks.

This technique could be incorporated into PATN by replacing the sequential refinement loop with a *net refinement* cycle (figure 14). Instead of SEQ organizing the subgoals into a sequential

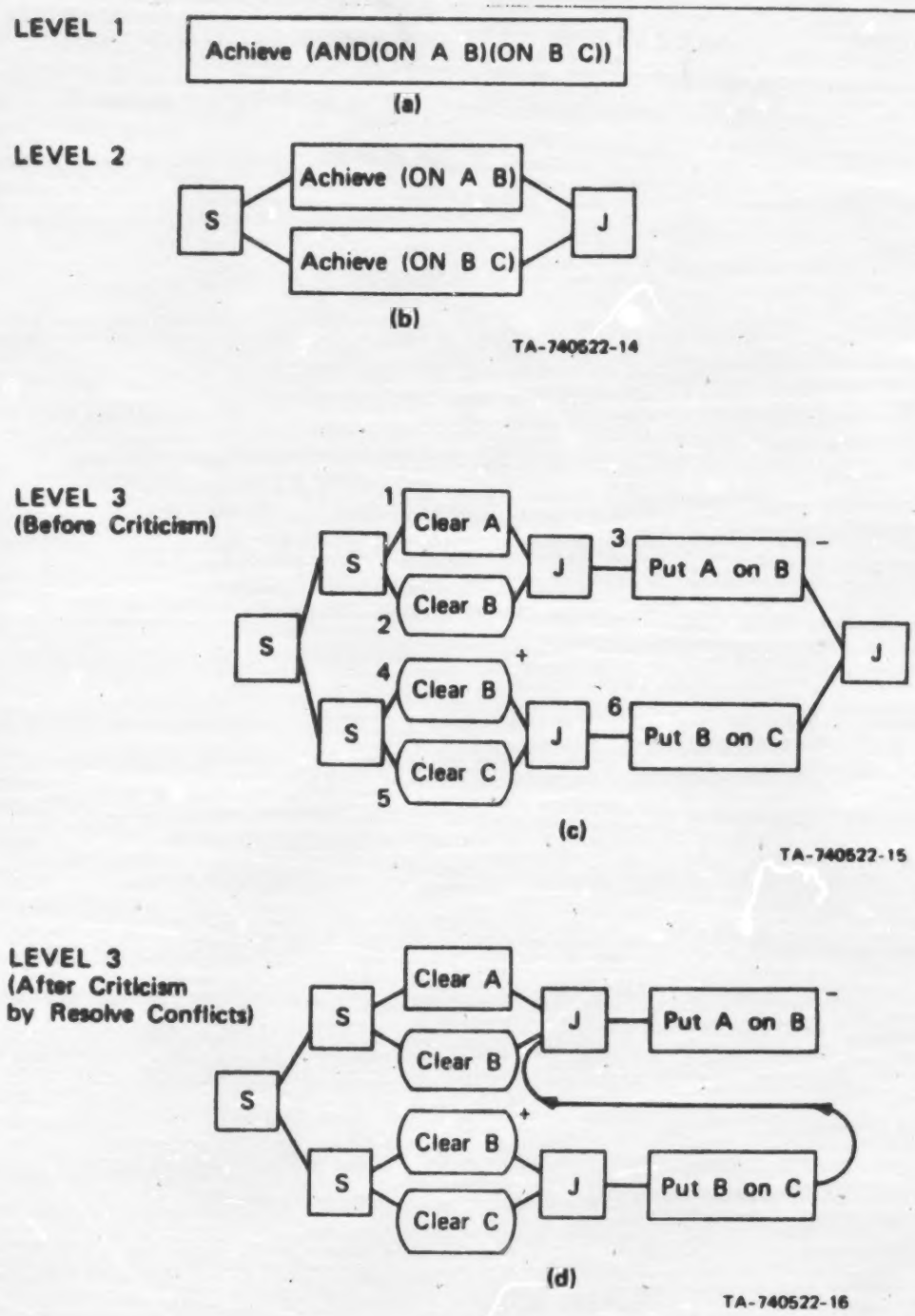


FIGURE 13
SUCCESSIVE REFINEMENT OF A PROCEDURAL NET FOR BUILDING A TOWER
FROM [SACERDOTI, 1975, P. 15]

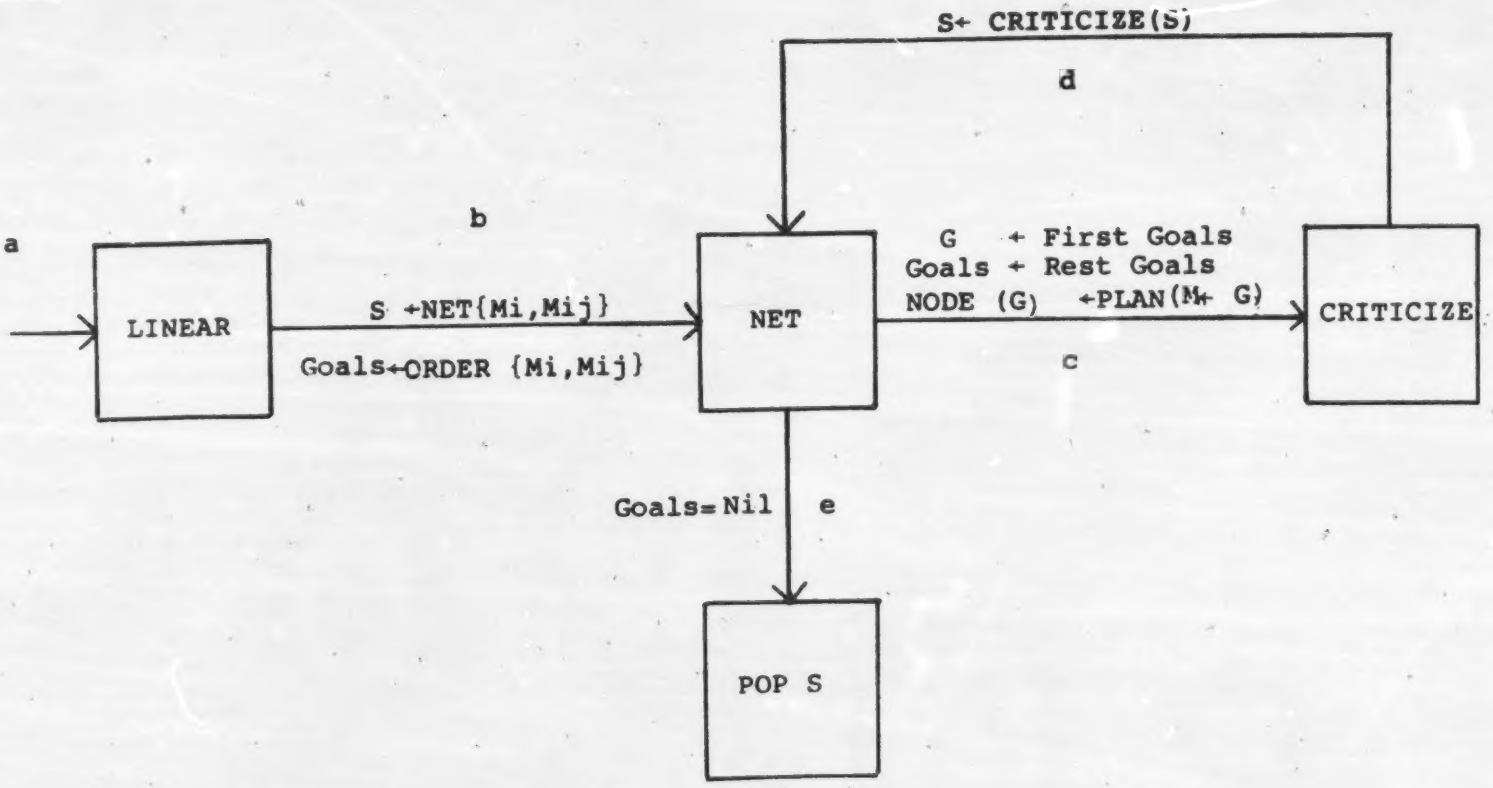


FIGURE 14

NET REFINEMENT

procedure, NET would organize the subgoals into a procedural net. This leads to a generalization of Sacerdoti's approach. We would represent in the net, not only the main subgoals as alternative branches unless ordering is required, but also the relations between these goals. Figure 15 shows the general form for such a "Structured Planning Net."

PATN would solve for each subgoal, following the procedural net technique of node expansion. But eventually the planner would also solve for the relations. When all subgoals, both to construct individual objects, and to satisfy their relations, were satisfied, the result would be an executable net. Any remaining branching could be executed in arbitrary order. Figure 14 illustrates this process. The operator NET on arc b sets the solution variable S, not to a superprocedure, but to a net of the form given in figure 15. GOALS is the set of subgoals, ordered for planning attention in the same fashion as for sequential refinement.

Arc c recursively calls the planner to solve for a subgoal. If the subgoal is a mainstep, it is spliced into the net as a refinement. But if the subgoal is a set of relations, then its solution may involve establishing a specific interface. If so, a sequence is enforced on the mainsteps adjacent to this interface. The effect would be that, in figure 16, A is transformed to figure B. If there are no relations between two mainsteps that require interfacing, then no additional ordering will be imposed and the net will preserve its branching. The result would be executable under the interpretation that parallel branches may be executed in any order. If there are a great many relations, then the net will ultimately reach its most constrained form -- a sequence.

Following Sacerdoti, Arc d would *criticize* the procedural net, checking for interactions that became apparent only after expansion. A typical example is noticing that the prerequisites of one subgoal are "clobbering" a brother subgoal. For the blocks world, this involves observing, by means of a table of *multiple effects* [Sacerdoti 1975, p. 209], that the prerequisites of one goal are clearing a block that was placed by another goal. We shall not go into detail regarding these critics. The interested reader should consult [Sacerdoti 1975]. However, it is worth noting that if

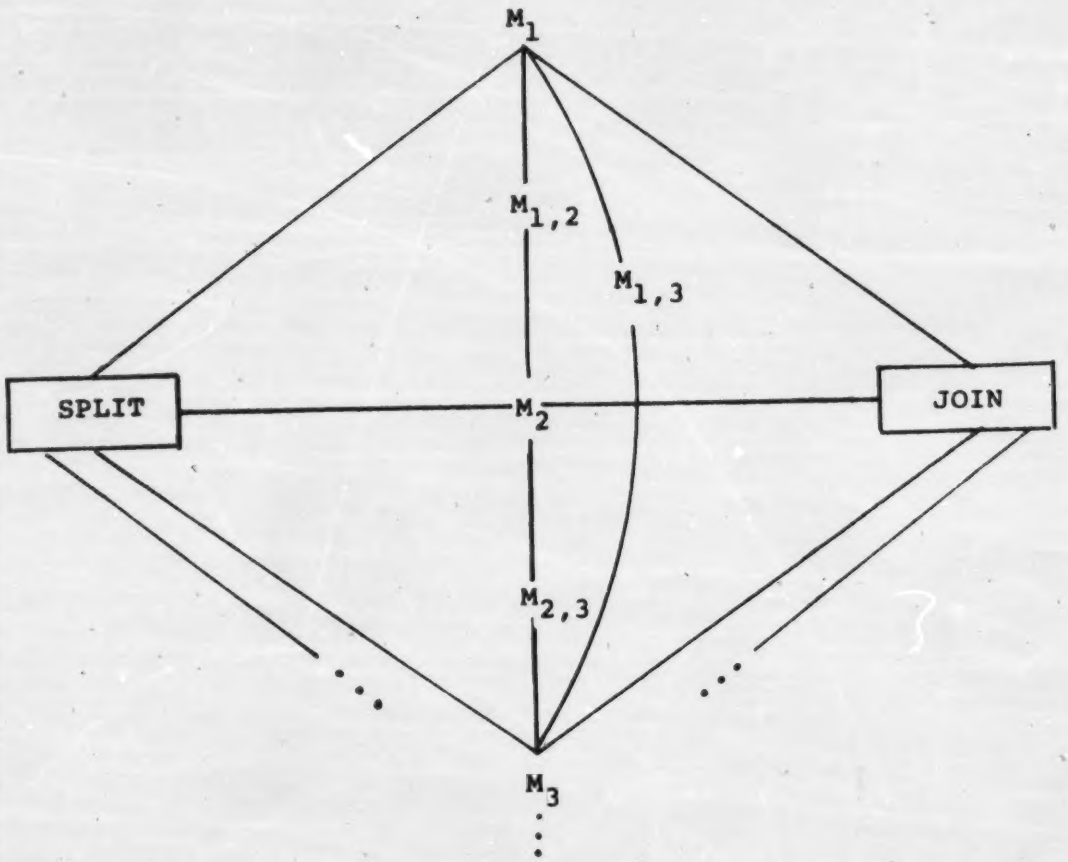


FIGURE 15
GENERALIZED PROCEDURAL NET

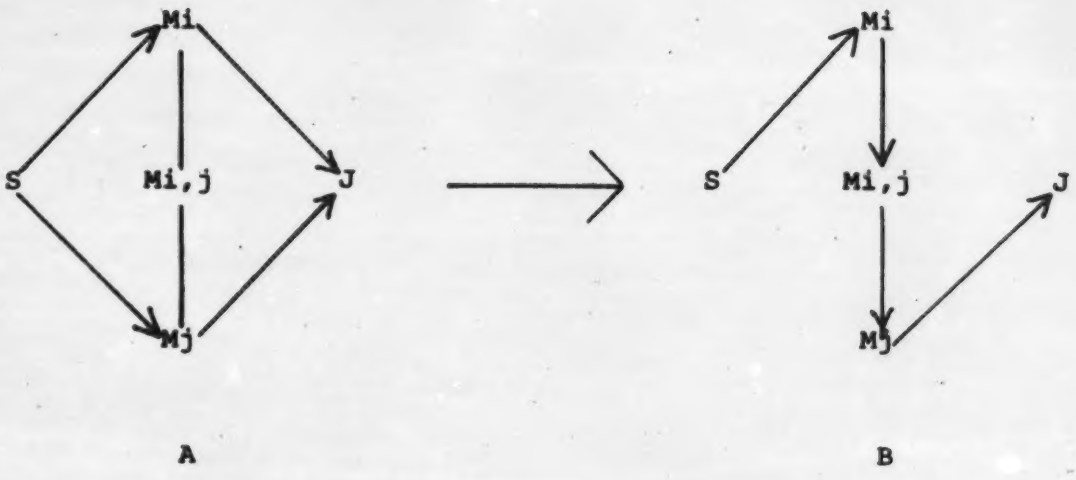


FIGURE 16
SOLVING FOR RELATIONS

the original linearization was completely successful, criticism should find no hidden interactions. But it is probably a useful heuristic check on the linearization cycle to include this criticism process.

There are subtleties in handling relations between non-adjacent mainsteps. For such cases, a relation such as R(X1,X3) might have to be replaced by an equivalent description in terms of objects accomplished by adjacent mainsteps, say (AND R_1(X1, X2) R_2(X2,X3)). We shall not discuss this further here. Our purpose here is only to indicate the direction our research would take in linking the ATN representation for planning concepts to Sacerdoti's procedural net representation for programs.

PATN's design represents an extension of NOAH, Sacerdoti's program for refining procedural nets, in that NOAH's primary planning technique is successive goal expansion. This corresponds to PATN's decomposition-by-conjunction. But PATN also represents a variety of other planning strategies, including repetition and the major category of reformulation. NOAH improves the representation of the procedures produced (by using nets), but does not emphasize PATN's central concern of how this goal structure is arrived at. Hence NOAH makes an important contribution, for the fashion in which it captures the principle of least commitment; but it is not a total theory of program composition.

3.3. Differential Diagnosis

Differential diagnosis refers to a collection of strategies which gather specialized selection knowledge at crucial choice points. Critics belong in this category. Critics analyze the problem description, and advise PATN as to which transitions are permissible and which are prohibited. A Block's World example is HACKER's critic (which could be attached to PATN's CONJUNCTION node), that diagnoses (AND (ON X Y) (ON Y Z)) problems as involving non-linear relations between the subgoals.

3.4. Lemma Libraries

Whenever a sub-problem is successfully solved, it can be added to the answer library, even if the overall approach fails.[14] This allows the problem solver to avoid repeated attempts to solve the same subgoals. Strips [Fikes et al. 1972] used *macrops* and *triangle tables* to achieve similar economies. This planning technique is analogous to the use of well-formed substring tables [Kuno 1967; Woods et al. 1972] in applying ATN's to natural language parsing, including their generalization to *charts*, as utilized by Kay [1973] and Kaplan [1973].

In the remainder of the paper, we consider the *rational bugs* that can arise in PATN's planning and how they can be diagnosed and repaired.

4. Structured Debugging

> Let us focus on one particular component of [general heuristic knowledge]: the art and techniques of ... *debugging*. The school experience is dominated by the normative attitude implied by "right answer vs. wrong answer". The mathematician's experience of mathematics is dominated by the purposeful-constructive attitude implied by the struggle to "make it work". He abandons an idea not because it happened to go wrong, but because he has understood that it is unfixable. Dwelling on what went wrong becomes a source of power rather than a piece of masochism (as it would appear to most fifth graders in traditional math classes).
>
> Papert, The Uses of Technology to Enhance Education, p. 10

We agree with Papert in his assessment that debugging is an essential part of problem solving. A powerful debugging system frees the planner from the necessity of always producing entirely correct plans. Bugs arise from heuristic choices made in constructing the plan. From the Structured Planning and Debugging standpoint, such heuristics are embedded in the default ordering of transition arcs. In the absence of specific arc constraints, PATN would prefer linear to non-linear plans, round repetitions to recursion. Such heuristics can lead to bugs. But we also expect these heuristics to provide several significant advantages to the planner, such as:

(a) allowing the planner to attempt new problem types with which it has had no experience;

(b) often being successful (because the default choice happens to be correct);

(c) in those cases where an error arises, regarding the nature of the difficulty as a specific diagnostic as to the locus of the incorrect decision and the alternative choice required;

(d) should subsequent experience lead to bugs, abstracting the problem description, embedding it in a critic at the point in the planning ATN where the incorrect choice was made, and thereby preventing future occurrences of the same error.

We call the class of mistakes that arise from reasonable heuristic judgments made in planning *rational bugs*. In this section, we show how this class of difficulties can be explained with reference to the planning theory. We introduce strategies for *Structured Debugging*, i.e., techniques

for diagnosis and repair of rational bugs, based on identifying incorrect or incomplete plans. We organize these strategies as a design for a *Debugger for Annotated PRograms (DAPR)*.[15]

For DAPR, debugging consists of diagnosis and repair. If we envision repair knowledge being associated with various classes of error, then once the underlying cause is identified, correcting the program is straightforward. Hence, the critical problem is diagnosing the underlying cause of the bug from its surface manifestation. We define a bug as being *manifest* if the program produced by a plan fails to satisfy the problem specification or model. The model consists of a Boolean combination of predicates over a set of objects: unsatisfied predicates are termed *violations* (following Goldstein [1974]). This definition subsumes the special case in which a program fails to run to completion due to an unsatisfied prerequisite of a primitive operation, since operators have Pre and Post models.

In terms of the ATN planning theory, the underlying cause of a bug is either an *incomplete* plan, in which a step is missing (e.g., the sequential refinement loop has failed to identify a subgoal), or an *inappropriate* plan, in which an incorrect arc transition has been made. Underlying causes can also be categorized as *syntactic*, *semantic*, or *pragmatic*, according to whether the malfunctioning planning knowledge lies in the topology of the ATN, the semantic arc constraints, or the pragmatic selection criteria (e.g., critics), respectively. (For additional details on this aspect of the bug taxonomy, the reader is referred to [Miller & Goldstein 1976c], in which these distinctions are made with respect to a context free grammar mirroring the topology of the ATN.) DAPR's goal in diagnosis is to identify where in the planning process an incomplete or incorrect choice was made.[16]

DAPR is designed to employ three diagnostic techniques: model, process, and plan diagnosis. Process diagnosis is the traditional kind of program analysis in which the programmer examines the state of the process at the point where the error is noticed. Model diagnosis goes beyond traditional programming environments and draws upon the formal specifications defining the

purpose of the program. Hence, it is a natural extension of work on verification. Plan diagnosis is new. It is made possible by a *derivation tree* being associated with the program, which represents the planning decisions made in creating the code. A diagnostic technique we shall not discuss that is useful in analyzing human code, but not especially appropriate for programs written by PATN, is code diagnosis. This amounts to having a list of rational form criteria, and examining the code to find if any are violated [Goldstein 1974, pp. 137-138]. As currently designed, PATN's set of planning techniques would not lead to this kind of mistake.

4.1. Model Diagnosis

Model diagnosis is the basic diagnostic technique, in that it involves the determination of whether the program has succeeded or failed in accomplishing its intended model. In logistic terms, it amounts to a verification in which the model predicates are applied to the structures -- pictures or block arrangements -- produced by the program.[17]

The particular set of model predicates which are violated provides strong evidence regarding whether the underlying cause is an incomplete plan: this is determined by checking if any code was generated whose purpose is to accomplish those predicates or their prerequisites. If the plan *is* incomplete, then repair can be accomplished by invoking the planner to supply the code.

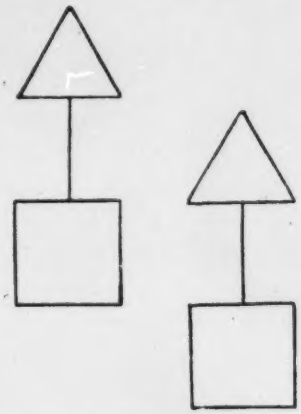
For example, suppose that after solving the wishingwell problem, PATN is asked to generate code for a scene consisting of two wishingwells, as shown in figure 17. This scene might be specified by the following model:

```
MODEL WW-SCENE
1 PARTS WW1 WW2
2 WISHINGWELL WW1, WW2
3 RIGHT-OF WW2 WW1
4 PARTLY-BELOW WW2 WW1
END
```

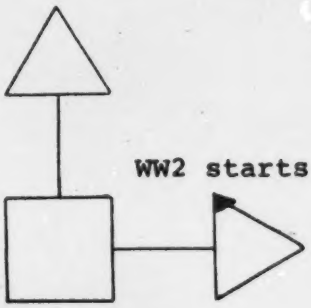
Both wishingwells would be accomplished by identification, that is, by calls to the existing subprocedure. PATN would initially generate a plan for this problem corresponding to the

FIGURE 17 - DEBUGGING A WISHINGWELL SCENE USING MODEL DIAGNOSIS

Intended Picture



Actual Picture



Manifest Violations:

WW2 does not satisfy the wishingwell model, because the bottom side of the roof is not horizontal.

Cause of the Bug:

The plan is incomplete. There is a Missing Prerequisite for this runtime environment. Wishingwell incorrectly assumes that turtle starts out facing north.

Repair Technique:

Use imperative knowledge of violated predicate (horizontal) to compute missing initial rotation.

following code:

```
TO WW-SCENE
10 WW
20 PENUP
30 FORWARD 100
40 PENDOWN
50 WW
END
```

Lines 20, 30, and 40 constitute an interface to accomplish model assertions 3 and 4. This code has a bug: the second wishingwell does not correspond to the wishingwell model, because the ROOF is not HORIZONTAL. Model diagnosis determines that, in fact, code exists in WW to accomplish this property. However, the plan for this code implicitly assumes that the turtle starts out facing NORTH. No code was generated to accomplish this prerequisite, since in previous uses of the procedure it happened to always be satisfied in the initial environment. Hence the underlying cause of the bug is *incomplete planning* arising from an unexpected runtime environment. The repair technique is to use imperative knowledge associated with the violated predicate to compute the missing code: an interface rotation step.

```
EDIT WW-SCENE
45 LEFT 90
END
```

Model diagnosis can succeed in cases such as this, where some predicate can be found for which no code exists to accomplish it. Alternatively, if the plan indicates that code *was* created to accomplish every applicable predicate, then further diagnosis is necessary. Perhaps there are unexpected interactions. Process diagnosis is the next stage.[18]

4.2. Process Diagnosis

Examining the state of the execution process, at the point where the bug became manifest, is often helpful in diagnosing unexpected interactions. This is the diagnostic technique used by HACKER. Conflicts between goals are diagnosed as non-linearities and reflect the underlying bug of having applied an inappropriate (i.e., pragmatically incorrect) plan. The essence is observing that one goal has violated a model predicate describing the intended effects of a prior step. The HACKER bugs of Prerequisite Clobbers Brother Goal, Strategy Clobbers Brother, and Prerequisite Conflicts with Brother are all of this type.

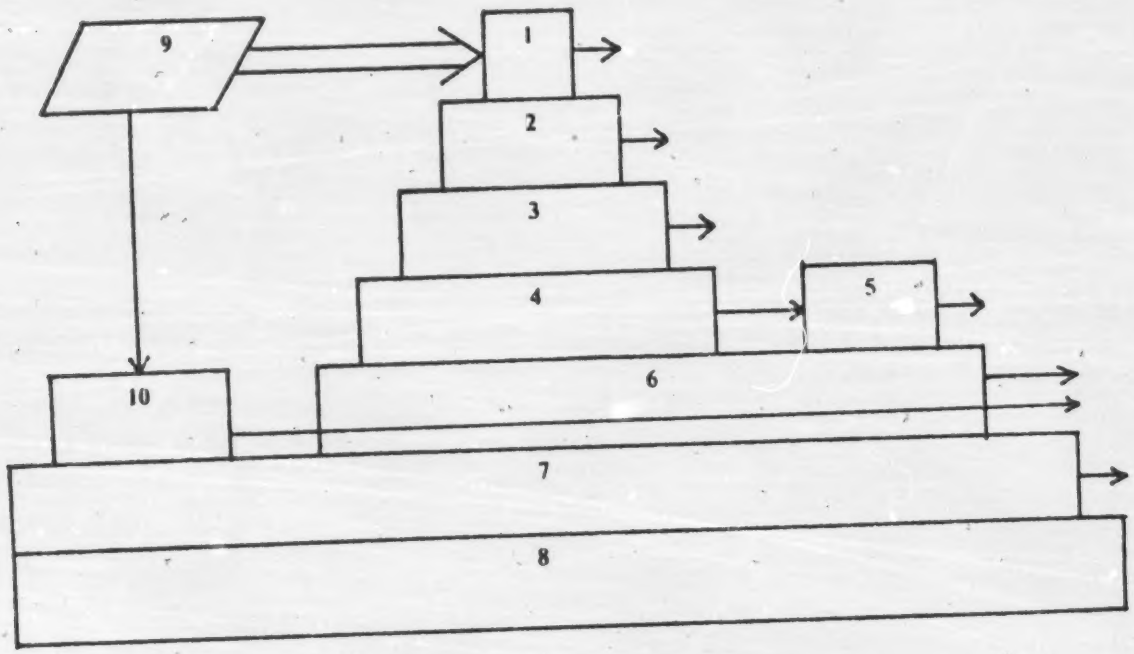
Sussman [1973] develops elaborate process state patterns for classifying kinds of interactions which we shall not repeat here. The essence is observing that a model predicate is being undone within a scope during which it is expected to be true. For example, consider the blocks world problem of building a tower of three blocks: (AND (ON A B) (ON B C)). Part a of figure 18 (from [Sussman 1974], pp. 10-11) diagrams HACKER's process state for a buggy first attempt on this problem. Each box represents a stack frame; the horizontal axis represents time; the vertical axis represents depth of procedure calls. This diagram matches the pattern (part b of the figure) for the bug type, *Prerequisite Clobbers Brother Goal*. Once the difficulty is thus classified, repair knowledge associated with that type of bug may be applied.

A predecessor of this diagnosis technique can be found in the PLANEX capability of the STRIPS problem solver [Fikes 1972]. In executing a plan, PLANEX checked for model predicates being accidentally undone. HACKER generalized this by checking for situations in which previously satisfied predicates are "intentionally" undone, i.e., where the plan itself is flawed.

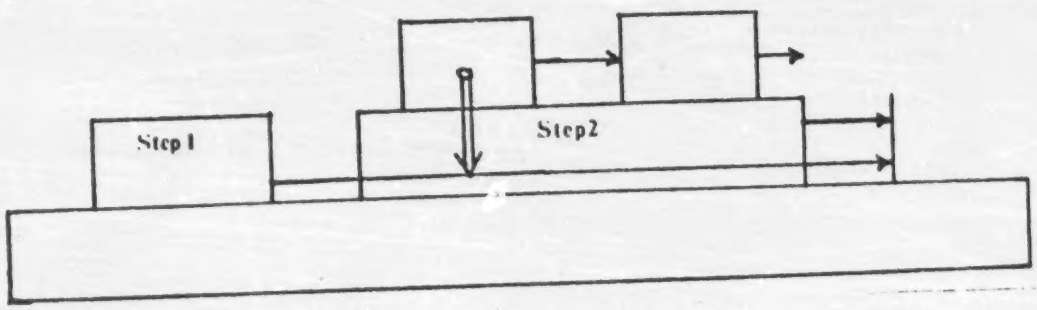
Process diagnosis can fail when the subgoal interaction is too complex for the debugger to recognize. DAPR would next resort to plan diagnosis, a new debugging technique not previously formalized, to aid in isolating the culpable design decision.

FIGURE 18

Debugging (AND (ON A B) (ON B C)) Using Process Diagnosis



a. Schematic Diagram of HACKER's Buggy Process



b. The General Pattern for PCBG Bugs

[from Sussman 1974, pp. 9-10]

4.3. Plan Diagnosis

Plan diagnosis is based on the fact that the planner has available knowledge of various heuristic decisions it has made which may prove unsuccessful. Associated with each node of the derivation tree for a PATN plan would be a specification of the values for a set of semantic variables. The values of these semantic variables correspond to "snapshots" of the contents of the ATN's registers at the time that the node was generated. The CAVEATS variable is the repository for advice regarding heuristic planning choices, for use in plan diagnosis.

Were PATN to decompose a model linearly, for instance, without any actual proof that no interactions existed, that fact would be recorded in the CAVEATS variable associated with the appropriate node of the derivation tree. Of course, such a simplification may turn out to have been incorrect. Consider, as a specific example, the task of drawing a face on the basis of the following model.

> *A FACE consists of two EYEs, a NOSE, a MOUTH, and a HEAD. (The two eyes are called LEFT.EYE and RIGHT.EYE.) The HEAD and EYEs must be CIRCLEs. The NOSE must be an equilateral TRIANGLE. The MOUTH must be a LINE. The EYEs, NOSE, and MOUTH must be inside the HEAD. The EYEs are to be above the NOSE. The MOUTH should be below the NOSE.*

```
MODEL FACE
1 PARTS LEFT.EYE RIGHT. EYE NOSE MOUTH HEAD
2 CIRCLE (HEAD LEFT.EYE RIGHT.EYE)
3 EQUITRI NOSE
4 LINE MOUTH
5 INSIDE (LEFT.EYE RIGHT.EYE NOSE MOUTH) HEAD
6 ABOVE (LEFT.EYE RIGHT.EYE) NOSE
7 BELOW MOUTH NOSE
END
```

In the absence of specific critics (i.e., before PATN had learned of the peculiarities of INSIDE) PATN would design the eyes and the head independently. But if the head and eyes are all circles of the same default size, then satisfying the relation that the eyes should be inside the head will be impossible. A linear plan that solves for the main steps independently of the relations leads to a bug.

DAPR would localize this difficulty using plan diagnosis. The key step is noticing the existence of a caveat, stating that the linear treatment of the subgoals for EYES and HEAD was justified only on heuristic grounds. In the absence of other guidance, this signals a potential bug. A closer investigation of the semantics of INSIDE would indicate a non-linearity with respect to the size property, which would then be recognized as the source of the problem: failure to observe a relevant pragmatic arc constraint on exit from the CONJUNCTION node (due to prior ignorance of that constraint).

PATN need not continue to make such mistakes in the future. Future performance could be improved by associating a critic with the conjunction plan node of the ATN. Thus, in subsequent problems, if two parts were described by the INSIDE relation, non-linear planning would be chosen immediately. In particular, the model would be modified to impose size properties on the parts so that, in terms of the revised problem description, linear decomposition would then be possible.

Caveats for use in plan diagnosis would also be generated when heuristics are employed during problem *reformulation*. The planner might construct what it believes to be an equivalent problem statement, but not in fact rigorously prove the equivalency. For example, two problem descriptions might be equivalent only over a subset of the possible inputs, but the planner might postpone determination of whether inputs outside of that range are ever possible or allowable. Such an heuristic approach, though frequently successful, can cause trouble. Hence, this too is recorded in the plan derivation and potentially noticed during debugging by plan diagnosis. In the case of allegedly equivalent reformulation, the CAVEATS variable associated with the equivalent-reformulation node of that derivation tree would contain the warning that the reformulation relied upon heuristic assumptions which were not rigorously demonstrated.

When such warnings are noticed during bug localization, DAPR's action would be to call upon more thorough analytic techniques -- such as formal demonstration of equivalency -- to see if

the heuristic assumption involved was in fact incorrect, thereby leading to an inappropriate plan. Some critics could involve such costly processing of the problem specification that, even though already learned from prior encounters, they might not be applied during initial planning. If plan diagnosis points to a possible error, these critics could subsequently be invoked.

4.4. Repair

DAPR's overall repair strategy for buggy PATN plans, once the culpable decision has been localized, is to undo the faulty choice and resume planning from that point. Selection of an alternative arc transition would be facilitated by procedural knowledge associated with:

a. the violated model predicate;

b. the bug type;

c. the plan type;

d. code caveats such as rational form criteria.

Some of this knowledge is domain specific (primarily knowledge of repair techniques for model predicates: Goldstein [1974] characterized knowledge of this kind for the Logo world.) The remaining knowledge is of the sort suggested in the discussions of the respective bug and plan types. For example, one repair strategy for a faulty equivalence reformulation, which failed to take into account the full range of inputs, is to design a conditional plan which separates the equivalence-preserving and non-equivalence-preserving inputs, and then to supply a separate solution for the non-equivalent case as well.

4.5. Limitations of the ATN Theory of Bugs

There are, of course, other kinds of bugs that arise in human programming that do not fall under the heading of rational planning errors. These range from execution errors to the construction of irrational plans. Execution bugs consist of those errors due to mistypings, misspellings, incorrect programming language syntax, noise on the computer line, and other such failures to successfully execute a statement of code. They are often diagnosed by the conventional computing environments, simply as a result of the code being unrecognizable. Repair is accomplished by correcting the side effects, if any, of the erroneous command, and then re-executing an edited version of the line. The plan is not affected.

Irrational plans can be precisely defined with respect to PATN. They correspond to making transitions that are not allowed in the planning network or failing to make transitions that are required. An example would be pursuing a repetition plan and failing to handle the terminal cases. PATN, as a theory of rational planning, does not explain these kinds of errors, and we shall not discuss them further here. (However, some potential implications of this distinction for structured programming are touched upon in the concluding section.)

Another source of dissatisfaction with programs (which we mention for completeness but do not pursue) arises from efficiency considerations. The Planning ATN is not a compiler and does not attempt to optimize the programs which are produced. As outlined here, DAPR would be restricted to correcting programs that fail to achieve their specifications. Programs that are far from optimal, but are nevertheless successful in terms of their models, are correct with respect to rational bugs. However, an interesting question for future research is to explore the extent to which PATN-like hierarchical annotation could provide guidance to an optimizing compiler.

In the next section, we elaborate the Structured Debugging approach to categorizing, diagnosing, and repairing rational errors, by analyzing the debugging behavior of HACKER, Sussman's [1973] blocks world problem solver.

5. Reconceptualizing HACKER

> ... the current bug classifier in HACKER is an *ad hoc* program and thus, the body of knowledge (called Types of Bugs in the overview flowchart) on which it operates is difficult to separate out and display. This, of course, makes Types of Bugs also very difficult to extend. The hope is, however, that Types of Bugs is essentially independent of the problem domain and need only be expanded when new problem solving methods (the Programming Techniques Library) are introduced. An important area for development of HACKER-like problem solving methods would be the systematization of the knowledge in Types of Bugs in a more modular way.
>
> Sussman, A Computational Model of Skill Acquisition, pp. 103-104

Sussman's HACKER program represents a landmark in AI theory for its emphasis on debugging as an important constituent of learning. However, HACKER is theoretically incomplete, in that it fails to integrate debugging expertise with a theory of plans. The underlying bug types in HACKER appear as a miscellany of debugging knowledge with no underlying regularity. The classification algorithm that maps manifestations to causes is *ad hoc*.

We shall extend the HACKER paradigm by developing debugging knowledge in the context of a coherent theory of planning. From this vantage point, the underlying causes of bugs are seen as specific errors in plan synthesis. The types of causes follow straightforwardly from the possible failings in traversing an ATN: failing to make an arc transition (incomplete plans), or making an incorrect arc transition (inappropriate plans). For example, failure to generate code to achieve the prerequisite conditions for a primitive constitutes a *semantically incomplete* plan.

In this section, we analyze HACKER from the PATN standpoint. The purpose is to demonstrate how PATN provides:

(1) greater theoretical clarity, by means of a unified planning and debugging theory;

(2) greater depth and breadth, by means of natural extensions to HACKER's set of bug types and debugging techniques.

There are four bug types in HACKER: Prerequisite Missing, Prerequisite Clobbers Brother

Goal, Prerequisite Conflicts with Brother, and Strategy Clobbers Brother. We analyze each in turn.

5.1. Bugs Arising from Incomplete Plans

The HACKER bug type, "Prerequisite Missing," is a special case of incomplete planning. This bug commonly arises in situations wherein the accomplishment of a model predicate depends critically on the particular environment in which the procedure is executed. Sometimes failure to generate code to satisfy a prerequisite (because it will happen to be true already in the expected initial environment) will be recognized as such during planning, and recorded as a caveat. The issue of dependency on the initial state was discussed in [Goldstein 1974, pp. 85-88] in which ASSUMPTION commentary was used to record known dependencies between the program and its initial environment. For the blocks world, Sacerdoti [1975] used what he termed *phantom* nodes to represent goals which happen to be true in the initial state, but which would otherwise need to be accomplished.

A rational planner may not realize (or be prepared to take the extensive time necessary to deduce) all potential interactions between the model and every possible (or intended) initial environment. For example, a plan may be used because the Post Model in the answer library matches the problem statement; but the planner may not prove that all the statements in the Pre Model must be true for all run-time environments. Hence, the plan might not be complete with respect to a new environment. In this situation, debugging consists of modifying or extending the plan to satisfy the set of newly violated predicates.

During *careful evaluation*,[19] missing prerequisites are manifested by primitives generating complaints. In the blocks world, for example, the robot will complain if asked to move a block to a position that some other object already occupies, or to grasp a block whose top is cluttered. Analogous complaints are generated by Logo turtle primitives. Logo will complain if the turtle is

asked to move off the screen or if a turtle command is executed prior to the display being created by a start-display function.

A unified approach is possible, which subsumes both the complaints generated by primitives and the broader class of model violations (referring to a program's failure to accomplish its goals). This synthesis is obtained by the use of Pre Models. If a Pre Model is associated with each primitive, then unsatisfied prerequisites simply become model violations. For example, as explained in section two, the Pre Model for the HACKER operator *Move block X onto block Y* would contain the assertions:

```
(CLEARTOP X)              ;X must have a cleartop to be picked up.
(ON X OLD-POSITION)       ;X must be at some known old position.
(SPACE-FOR X Y) ;The top of Y must have room for X.
```

The inclusion of unsatisfied prerequisite manifestations in the class of model violations, and the classification of prerequisite missing bugs as semantically incomplete plans allows a unified treatment of diagnostic and repair techniques. Each model predicate, whether part of a primitive operator's Pre model or a problem's model, has procedural knowledge associated with it that aids in isolating the bug locus, proposing repairs, and thereby completing the plan.

5.2. Bugs Arising from Incorrect Conjunctive Plans

Prerequisite Clobbers Brother Goal and Nonlinear Composition

Prerequisite Clobbers Brother Goal (PCBG) and *Prerequisite Conflicts with Brother* (PCB) bugs both arise from a linear plan being applied to a non-linear problem. PCBG is the underlying cause when attempting to build towers incorrectly from the top down. In HACKER terms, the goal is (MAKE (AND (ON X Y) (ON Y Z))). HACKER's default solution is to achieve the conjuncts in the order in which they appear. That is, this bug arises in situations in which the planning system ignores the possibility that one conjunct may *have* to be accomplished prior to the other. From the PATN standpoint, this bug is caused by the planner following the (pragmatically inappropriate)

linear arc from the conjunction node. PATN's default, as explained previously, is to choose a linear plan except when non-linear composition or decomposition critics detect an interaction.

In these terms, it is clear how debugging is to be accomplished. DAPR would re-apply PATN to the problem with the advice that a linear plan is prohibited. This knowledge is represented as an NLC predicate on the arc from the CONJUNCTION node to the NONLINEAR COMPOSITION node.[20] (See figure 7.) The predicate checks for patterns of the form, (AND (ON X Y) (ON Y Z)), in the problem model. If they occur, planning control is transferred to the NONLINEAR COMPOSITION node, with composition guidance being appended to the ADVICE register. This advice, computed by the NLC predicate, directs the order of re-composition when planning eventually reaches the Sequential Refinement loop. (See the overall ATN flowchart of figure 2). For the tower example, the effect of the advice is to ensure that the plan for achieving (ON X Y) is executed *after* the plan for (ON Y Z).

Sussman analyzed these bug detection patterns, but had no coherent place for them in an overall theory. From the standpoint of an ATN planner, they represent constraints on arc transitions, and their effects are to set registers to guide subsequent planning.

Prerequisite Clobbers Brother and Nonlinear Decomposition

PCB arises in the following problem: HACKER is asked to find space for both blocks A and B on base block C, i.e., to accomplish figure 19b. In attempting this problem linearly, HACKER first places A on the center of C (figure 19c), with no consideration of the brother goal of placing B on C. When the time comes to place B on C, there is insufficient room and block A must be pushed left (figure 19d). This results in a Double Move (rational form) manifestation. HACKER's debugging strategy is to construct a plan that simultaneously takes account of both prerequisites: (PLACE-FOR A C) and (PLACE-FOR B C).

The PATN-DAPR approach is to have the debugging episode produce a non-linear

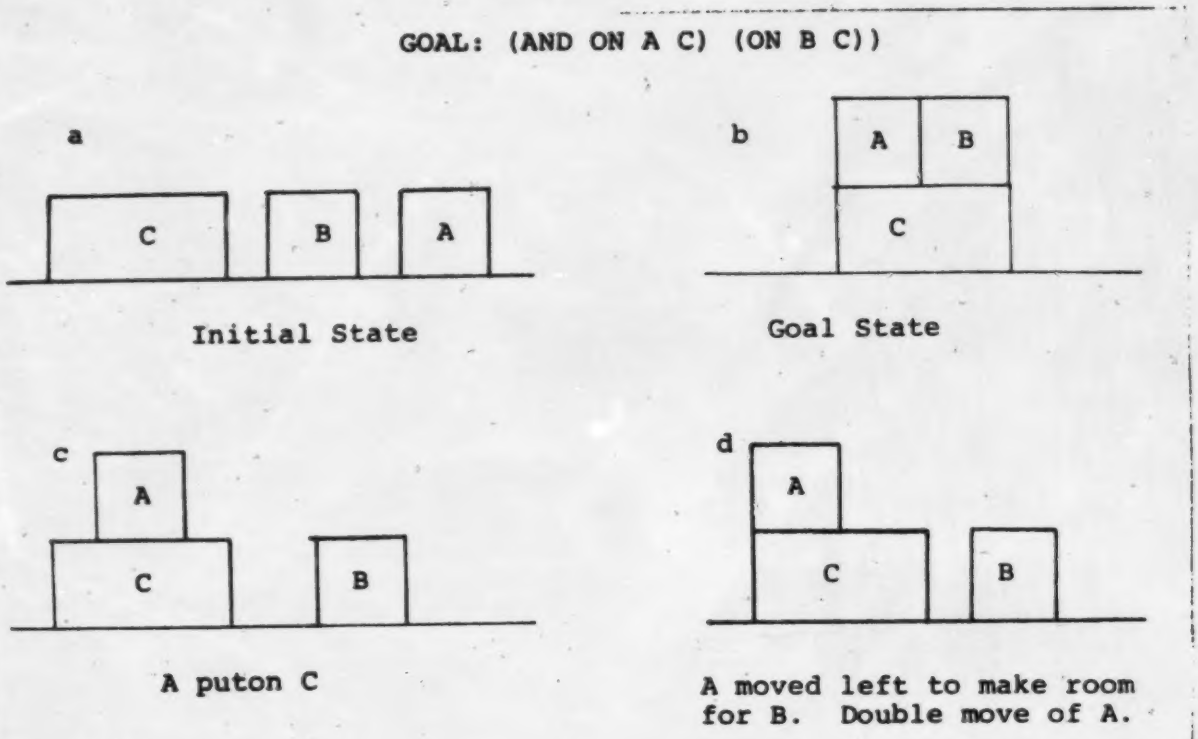


FIGURE 19
PREREQUISITE CONFLICTS WITH BROTHER

decomposition critic that triggers on multiple SPACE-FOR predicates: (SPACE-FOR X Z), (SPACE-FOR Y Z). After triggering, the critic's action is to append to the problem description register, M, location predicates for X and Y with respect to Z. Given explicit locations, a linear decomposition can take place.

PATN does not go beyond HACKER in handling this difficulty. The only claim here is that the ATN representation helps in understanding the issues involved. The planner's classification of conjunctive non-linearities into non-linearities in the decomposition or in the composition (e.g., their order) makes both PCBG and PCB understandable -- and even to be expected -- given a default preference for linear plans.

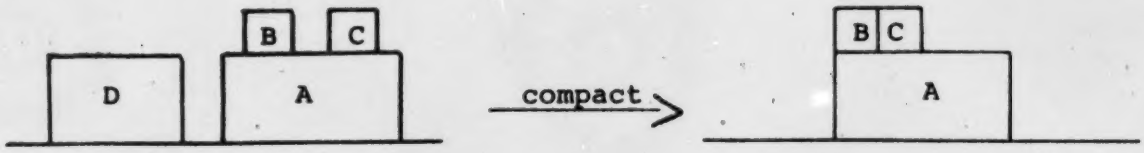
5.3. Bugs Arising from Incorrect Disjunctive Plans

The bug *Strategy Clobbers Brother* (SCB) arises when two different strategies are attempting to accomplish the same goal, but conflict with each other. The particular blocks world example discussed in HACKER involves the *findspace* strategy "remove block from surface" conflicting with its brother strategy "compact by pushing to the left" (figure 20). Removal can undo a prior compacting. HACKER notices the conflict and debugs by imposing an ordering on these strategies. Removal ought to be accomplished before compacting.

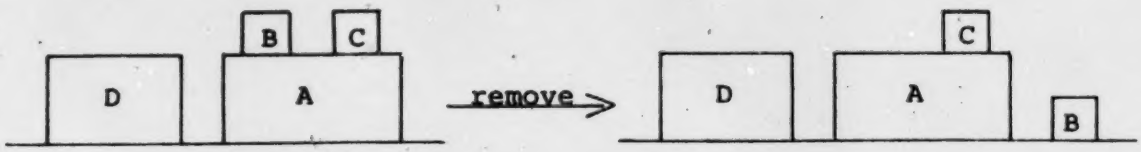
SCB can be understood in PATN terms as arising from an incorrect transition at the node for disjunction plans. The disjunction is in the set of alternative strategies for accomplishing the FINDSPACE goal. Although disjunction plans were not covered in section two, extending the basic PATN design to handle this additional logical operand is not difficult. Figure 21 illustrates a planning taxonomy for the decomposition of disjunctions.

The first major decision involves resolving whether the disjuncts are exclusive or additive. Exclusive disjunction refers to a set of options in which only one can be chosen. Exclusive disjuncts cannot "partially" succeed. Crossing the Atlantic by steamer or plane are mutually

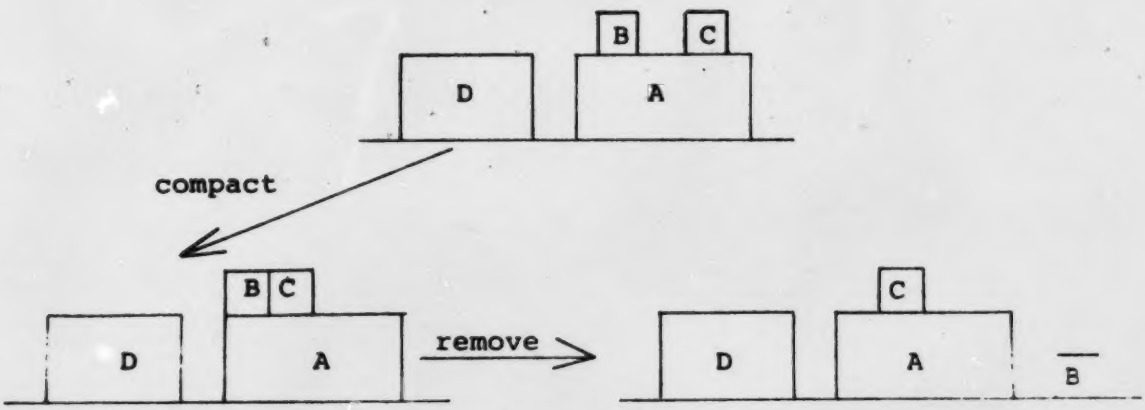
GOAL: (AND (ON C A) (ON D A))



Compacting: Blocks pushed to leftmost position



Removing: Blocks not required on A are removed



Top of A is compact

Conflict: Top of A no longer compact

Compacting then removing leads to conflict. The removing strategy has undone the compacting.

FIGURE 20
STRATEGY CLOBBERS BROTHER

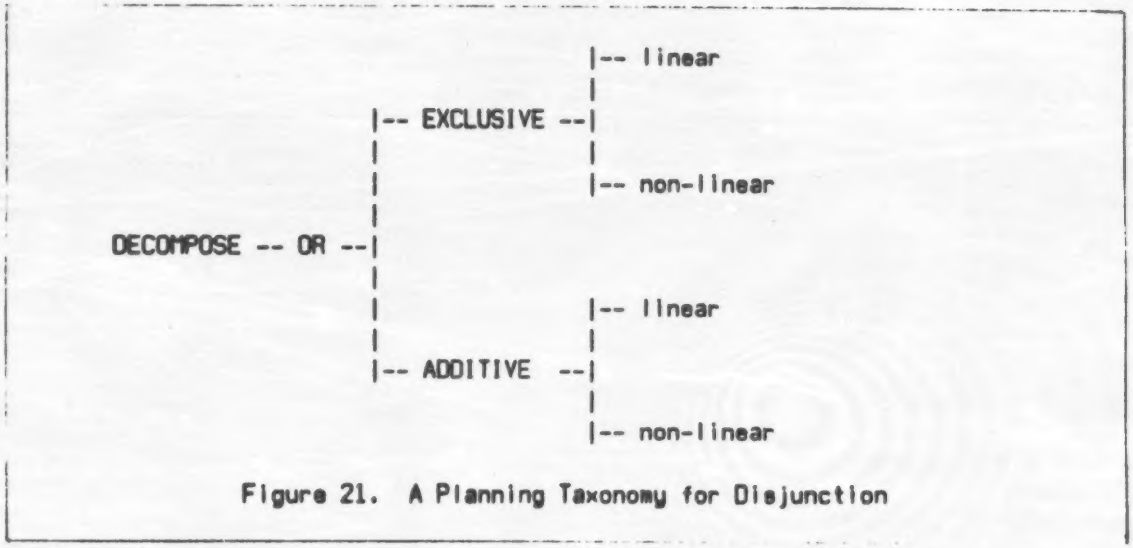


Figure 21. A Planning Taxonomy for Disjunction

exclusive travel strategies. One does not travel half way by plane and then switch to ship.

Additive disjuncts can partially succeed and indeed may behave cooperatively. Strategies for finding space are of this kind. However, after deciding that the disjuncts are cooperative, the question of whether there are possible interactions is still open. We intend to implement this in PATN in a similar fashion to the handling of conjunction, with linearization cycles.

Relative to this taxonomy, the underlying cause of SCB is an inappropriate arc choice, similar to PCBG and PCB. The difference is only that the nonlinearity which has been ignored is relative to alternative disjuncts, rather than conjuncts. The planner may have chosen, by default, to treat subgoals as independent additive disjuncts, when in fact they are dependent: subject to a non-linear constraint on their order of application. The appropriate debugging techniques are also similar, with corrective knowledge being attached to the arc transitions out of the DISJUNCTION node of the ATN.

5.4. Generalizing the HACKER Paradigm

This section has argued that analysis of the faults in plans as incomplete or inappropriate arc transitions provides a unifying framework in which to understand the miscellany of HACKER bug types. We conclude this aspect of the discussion by summarizing the dimensions along which PATN allows a broader view of program planning and debugging than is present in HACKER.

1. HACKER contains an implicit theory of planning, consisting of an assortment of programming techniques. A program is written through successive macro expansion using these techniques. We think that the PATN framework surpasses HACKER along this dimension, bringing greater organization to planning. Rather than as a "bag of tricks" [Sussman 1973, p. 57], PATN would organize programming knowledge as decomposition techniques that convert the standard logical operators -- AND, OR, FOR-EACH -- into procedural form. From this standpoint, HACKER's program writing capability is a subgraph of the planning ATN, consisting

of the identification and decomposition portions, but excluding problem reformulation.

2. HACKER is critically dependent on the annotations associated with the programs it writes; but no clear theory of annotation is present. The linguistic analogy underlying PATN leads to a concept of program annotation as the hierarchical derivation tree that the ATN generates, augmented by semantic variables associated with nodes in the derivation tree (which specify such contextual information as the problem specification, debugging caveats, and re-composition advice). The set of semantic variables available during debugging is not arbitrary or *ad hoc*, but corresponds to snapshots of the contents of the ATN's registers during planning. PATN's notion of commentary follows from the structure of its grammar, and from the semantics and pragmatics of its augmented transition network.

3. By having a comprehensive set of planning constructs, it is possible to predict additional types of bugs. For example, just as the wrong choice between linear and non-linear conjunction plans leads to bugs, so too does the wrong choice between any set of mutually exclusive planning arcs emanating from a given node. Thus, a similar class of bugs can be expected to arise in deciding between round (simple tail recursive) and fully recursive repetition plans; and, indeed, in human problem solving, this confusion is often displayed. Another class of bugs arises when one conjunct does not completely "clobber" another, but partly interferes. The potentiality for this is apparent when it is remembered that problem descriptions may be more complex logical models than those addressed by HACKER. An example of this in the blocks world is, "build two green towers." There may be no interference between the choice of color, but there may be interference in the choice of blocks, as would occur if only a limited number of blocks were available.

4. HACKER's *critics* can be characterized as transition constraints on ATN arcs. From this broader viewpoint, one immediately notices the possibility for positive as well as negative critics, which argue for or against particular plans. More generally, given the situation of choosing a transition arc out of a given state in the planning network, a critic is simply some selection function

on the arcs.

5. Unsatisfied prerequisite manifestations can be considered instances of the more general class of model violations. All that is needed is to include operator models as well as problem models. This is not an added burden, since operator models are necessary anyway as part of the primitive library used by the identification planning technique.

In concluding our discussion of HACKER, we must stress that we agree with the overall HACKER philosophy that problem solving consists of both planning and debugging. Our objection is that HACKER treats these two complementary activities in an isolated fashion. HACKER does not pay sufficient attention to the theory of description for problems, for operators and for plans. We have tried to illustrate how our linguistic theory of planning and debugging remedies this.

6. Conclusions

> The proper study of those who are concerned with the artificial is the way in which that adaptation of means to environments is brought about -- and central to that is the process of design itself. The professional schools will reassume their professional responsibilities just to the degree that they can discover a science of design, a body of intellectually tough, analytic, partly formalizable, partly empirical teachable doctrine about the design process.
>
> Simon, The Sciences of the Artificial, p. 58

In striving to achieve a rigorous, unified theory of planning and debugging, we have used concepts from computational linguistics to characterize the problem solving process. Planning concepts were represented using an augmented transition network, resulting in a structured theory of planning which appears to be both powerful and clear. Debugging was analyzed as the diagnosis and repair of incorrect or incomplete plans, which inevitably arise in the course of rational but heuristic planning. We conclude by summarizing the limitations, extensions, and potential applications of the Structured Planning and Debugging theory.

6.1. Limitations and Extensions of Structured Planning

> My mind was struck by a flash of lightning in which its desire was fulfilled.
>
> Dante, Paradiso (Canto XXXIII), in [Polya 1965, p. 54]

Of course, there are many aspects of human problem solving and its *flashes of lightning* that we have not touched upon. What follows is some of the specific limitations that we perceive in the theory embodied by PATN, and possible extensions to remedy them.

In section three, we discussed how the generation algorithm running over the ATN could be improved. These improvements could obtain better performance within the boundaries implied by the knowledge present in the network. They do not address those limitations inherent in the particular subset of planning knowledge present, i.e., the basic taxonomy.

Bearing in mind that our problem descriptions are composed of logical operators, it is readily apparent that the network currently contains techniques for solving conjunctions and universal

quantifications over a finite domain (repetitions); however, the network does not contain strategies for handling disjunctions, negations, or existential quantifications. These clearly could be incorporated using the ATN formalism, but we have not addressed the last two in this paper. (Disjunctions were briefly discussed in section five.)

Moreover, with techniques for all of the logical operators, the planner would still remain incomplete. Even if a problem is described as a conjunction, the planner may not find the constructive solution necessary to accomplish the conjuncts. Interactions might exist that make it impossible, or the particular technique for resolving a certain interaction may be unknown. Nevertheless, we believe that the logistic framework for describing problems at least gives a superstructure on which to build more elaborate planning techniques. The success of this superstructure can be evaluated by the exten. to which future research allows the collection of decomposition and linearization techniques to be extended within the ATN framework.

Another PATN limitation lurks in the *ad hoc* nature of its reformulation techniques. Theoretically, a general theorem prover could enumerate all equivalent models. But such a strategy would be computationally too costly to be useful. Instead, we enumerated a small number of heuristics. Future research might attempt to find a middle ground between general deductive strategies and specific procedural heuristics. Such an accommodation is suggested by recent work on theorem proving [Kowalski 1973; Moore 1975].

In designing PATN, we have emphasized an hierarchical approach to planning. Such a philosophy is a simplification in that it does not take account of possible heterarchy [Minsky & Papert 1974]. By this we mean that in some planning situations, a person clearly takes advantage of bottom up evidence to guide an ordinarily top-down analysis. Information and decisions do not inevitably flow in a single direction. A robot that trips over a bag of money on its way to rob a bank should not kick the money aside and continue with the caper. Figure 22 illustrates an Heterarchical Refinement loop, in which goals can be reordered after each recursive solution for a

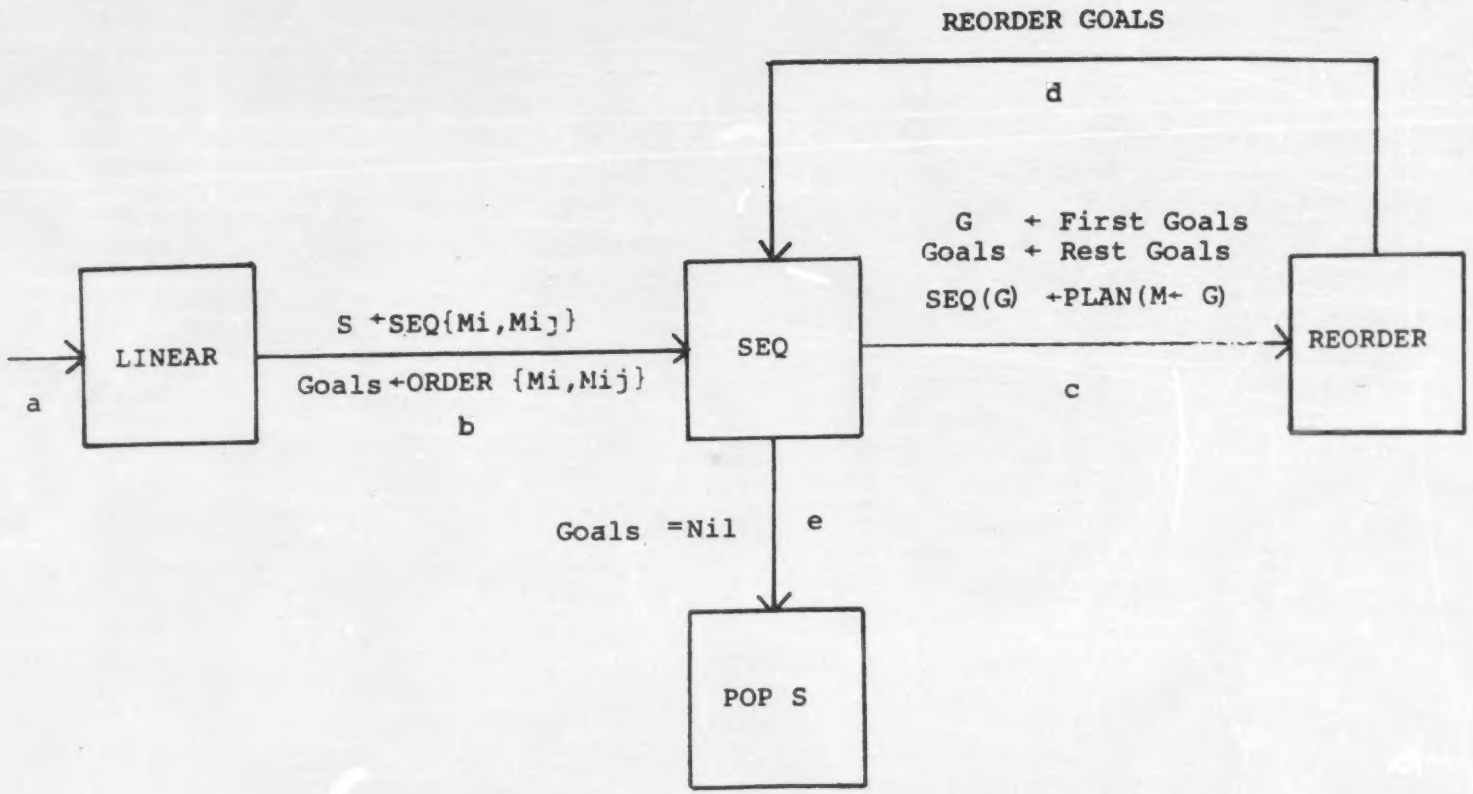


FIGURE 22
HETERARCHICAL REFINEMENT

subgoal. Eventually this sort of complexity must be addressed. However, our research plan is first to construct and experiment with a clearly top-down structured planner, in order to better understand its limitations as well as its virtues.

6.2. Summary of the Structured Debugging Viewpoint

In designing DAPR, bugs and debugging were analyzed in the context of the Structured Planning theory underlying PATN. Since PATN represents planning knowledge using an augmented transition network, it is possible to describe the underlying causes of bugs as specific classes of erroneous arc transition decisions during planning. The general form of a bug can either be failure to include a needed constituent, or inclusion of an inappropriate constituent. These failures can be caused by ignorance of or failure to obey ATN arc transitions and the constraints on those transitions.

DAPR's debugging consists of diagnosis and repair. These activities are characterized by the various data structures on which they operate. PATN employs four representations for a procedure: the problem description ("model"), the process ("chrontext"), the code, and the plan derivation. The theory provides a notion of annotation as derivation trees, which summarize the design decisions leading up to the actual code. This thorough, hierarchical representation of the history of the solution allows for a deeper analysis of debugging which we believe will be of practical value: for example, in the construction of programming environments.

The ideas in this essay have developed from those of Papert [1971a,b; 1973], Sussman [1973], and Goldstein [1974]. To provide perspective on its relationship to earlier work, the current theory was contrasted with Sussman's HACKER. The claim that the present approach subsumes that of HACKER was defended by several specific arguments. The relationship of HACKER's bug types to the current classification scheme was discussed.

In the remainder of this section, we describe various applications of the Structured Planning and Debugging paradigm: to protocol analysis, structured programing, and computer aided instruction.

6.3. Protocol Analysis

In [Miller & Goldstein 1976b], an earlier version of the planning grammar was applied to the task of parsing elementary programming protocols. The recognition process was performed manually, by the authors. Continuing our strategy of applying concepts from computational linguistics to problem solving, we plan to experiment with the application of various algorithms for natural language comprehension to the task of automated protocol analysis.

A critical question that arises is whether PATN provides a *spanning model* for elementary human problem solving. By this we mean: if PATN is put in a mode wherein it generates all possible solutions to a given problem (primarily through successive reformulations), will the set of programs produced include most of the successful solutions generated by people? More critically, can PATN's solution *process* -- at an appropriate level of abstraction -- mimic that undergone by human problem solvers? More specifically, is the protocol analysis task profitably approached from the standpoint of determining which of PATN's possible plans for a given problem is being used?

We do not know whether PATN will be sufficiently powerful to include all of the plans typically pursued by students in elementary Logo programming tasks. If so, it will represent a step forward in information processing psychology [Newell & Simon 1972].[21] Our preliminary analyses of many Logo protocols have been encouraging. But extensive experimentation is needed before a definitive answer will be available. Fortunately, we are in a good position to attack this set of psychological questions because the Logo project has collected extensive data on student performance [G. Goldstein 1973; Okumura 1973].

[Miller & Goldstein 1976d] presents a preliminary design for *PAZATN*, a PATN-based automatic protocol analyzer. In applying PATN to protocol analysis, we envision *modeling the individual* by inducing, from previously analyzed protocols, a *personalized* (modified) version of the Planning ATN. The success of these models will be judged by the extent to which they successfully predict subsequent behavior on the task. Again, experimentation is needed to determine whether this approach is viable.

The parsing problem is complicated in analyzing human protocols by the possibility of irrational planning errors and execution errors, in addition to the rational planning bugs discussed earlier. Because of the increased uncertainty introduced by possible mistakes in executing a statement of code or constructing an ungrammatical plan, we envision taking advantage of the powerful search procedures created for parsing speech utterances (such as those described by Allen [1975], Woods et al. [1975], Paxton & Robinson [1975], and Lesser et al. [1975]), in which uncertainty in the auditory interpretation similarly complicates the parsing process.

6.4. Structured Programming

> ... the new reality is that ordinary programmers, with ordinary care, can consistently write program segments which are error free from the start.
> Harlan D. Mills, "On the Development of Large, Reliable Programs," *Proc. IEEE Symp. Computer Software Reliability*, 1973, p. 155.

It is sometimes argued by proponents of structured programming that discipline in coding can eliminate all bugs. The Structured Planning and Debugging theory sheds some light on this issue. Rational bugs are unavoidable (or, at least, not worth avoiding). They correspond to heuristic planning judgments made when no better criteria were available, as often occurs when programmers are solving new problems. It is probably through the experience of whether their default heuristics succeed or fail on a new class of problems that individuals acquire skill. On the other hand, irrational errors and syntactic planning bugs must surely be increased by unstructured, careless programming. It is this class of errors, not rational bugs, that we believe the structured

programming movement as a whole has in mind, in calling for more disciplined planning and coding.

Hence, a potential application of our theory is to the design of improved environments for programming. In [Miller & Goldstein 1976c] we have presented the design for a programming editor called SPADE-0, which encourages articulate, structured planning, using a context free grammar. The virtues of working within such an editor, in which programs are specified in terms of their plans, include: (a) expressing one's programming ideas in this fashion can lead to increased clarity, by drawing the programmer's attention to the nature of the plan being applied; and (b) articulating the plan increases the system's leverage to aid in the diagnosis and repair of bugs.

However, context free grammars have limitations which prevent SPADE-0 from exceeding a certain plateau of utility. These limitations can be overcome by representing plans, not in terms of context free rules, but in terms of an ATN. Consequently, we envision using PATN to extend the capabilities of the SPADE editor, creating an improved version, SPADE-I. One might instruct this improved editor to change a particular subgoal from being accomplished by means of IDENTIFICATION to a plan based on DECOMPOSITION by CONJUNCTION. The reason might be that the original subprocedure fetched from the library had unacceptable side effects. SPADE-I, the PATN-based editor, could then lead the programmer through a sequence of top-down planning decisions that would realize the new plan. Because of the availability of PATN, SPADE-I could, among other improvements, assume greater responsibilities concerning low level coding decisions.

PATN is a top-down structured programmer. As a result, the SPADE-I editor could assist the programmer in exactly this process. The advantages of such an editor over conventional programming environments derive from a broader and deeper taxonomy of planning concepts. Thus, while we believe that Dijkstra and his colleagues have pointed in the right direction, in

calling for a structured approach to programming (see, e.g., [Dahl et al. 1972]), we also believe that the type of research involved in constructing PATN provides an essential next step: detailing exactly what rational planning involves.

In future research, we plan to construct the PATN-based SPADE-I editor, and to experiment with its performance as a programming tool. The criteria by which it may be judged are the extent to which programmers find it useful, and its effect on program writing and debugging time.

6.5. AI-based Computer Aided Instruction

In designing AI-based CAI programs, three critical problems are: (a) inducing a model of the student; (b) having a model of the expert; and (c) generating a tutorial plan for guiding the student toward expert competence. PATN may aid in the resolution of these three problems in the design of CAI systems for tutoring programming and problem solving.

We have discussed how PATN may provide an important modeling tool. Implicit in PATN is also a theory of learning. From the PATN standpoint, learning is the acquisition of new grammatical rules, new semantic variables, and new pragmatic constraints for deciding between alternative plans. Hence, a PATN-based tutor could compare the topology of the personalized ATN induced for the student to the full PATN grammar, and choose a difference as the issue to be taught. Alternatively, the tutor could parse a given protocol, compare it with how PATN would have solved the problem, and utilize the differences as the specific issues to be discussed with the student in analyzing his or her performance on the problem. For example, a pragmatic planning bug might be attributable to the absence of a relevant critic. In this fashion, we attempting to extend the *Issues and Examples* paradigm, developed by Burton and Brown [1976] for an elementary arithmetic world, to the more complex environment of programming and problem solving [Goldstein & Miller 1976a].

Of course, there are many other subtleties in designing intelligent computer tutors not touched

upon here, such as: (a) in what sequence should the knowledge be taught? (b) how intrusive should the tutor be? (c) how can the tutor's behavior be explained to the student, so that its actions are not mystifying? and, (d) can sufficiently powerful natural language capability be provided so that the student can interact comfortably with the tutor? Nevertheless, PATN is a necessary ingredient, as it provides a model of the planning expertise which is to be conveyed by the tutor.

It is also worth observing that automatic protocol analysis and student modeling, even without automatic tutoring, could be valuable to a human teacher. The parsed protocol and student model might allow the teacher to notice more easily when the student is relying on a limited lexicon of planning strategies, and whether the strategies that are known are organized in a successful fashion. This kind of detailed description of the reasoning process offers the possibility of escaping from the tyranny of standardized tests, whose outcome is an uninformative numerical score.

6.6. The Science of Heuristic

Polya has called *heuristic* the study of the "means and methods of problem solving" [1962, p. vi]. His various books [1957, 1962, 1965, 1967] offer insight into the nature of problem solving, discussing skills and abilities far in advance of the most intelligent AI programs. But heuristic, as Polya develops it, is not yet a science. There are no formal representations for problem solving concepts; no rigorous means for experimenting with alternative theories. The use of the computer to implement and experiment with such theories makes the study of heuristic a science. PATN represents a small contribution to this enterprise by experimenting with a particular procedural representation -- the augmented transition network.

The most common criticism of even the most insightful analyses of problem solving is -- "but how can I realize when a particular problem solving strategy is appropriate?" The gap that exists between informal, intuitive discussions of thinking, and specific, useful guidelines, is illustrated by

the self-description of the great mathematician Poincare cited in section two:

> Every day I sat down at my table and spent an hour or two trying a great number of combinations, and I arrived at no result. One night I took some black coffee, contrary to my custom, and was unable to sleep. A host of ideas kept surging in my head; I could almost feel them jostling one another, until two of them coalesced, so to speak, to form a stable combination.
>
> Poincare, H., "Mathematical Discovery," in [Rapport 1963, p. 132]

Surely we can do better than advising a student to drink coffee before going to sleep.

Attempting to structure the skills of various fields, whether mathematics or carpentry -- in a form that provides useful, precise guidelines to students -- is the fundamental task of education. Research in computer science, computational linguistics, and artificial intelligence is finding representations for active knowledge that are precise, powerful, and perspicuous. Ultimately, PATN's most important contribution is as an experiment in this vein: exploring whether a particular computational formalism is useful as a representation of problem solving skill. As such, it is a vital part of that investigation of the design process which Simon calls for in the quotation with which we began this section.

7. Notes

1. The name *Structured Planning and Debugging* emphasizes several themes. One theme is that the use of concepts from computational linguistics has been helpful to us in *structuring* our theory. Expressing a cognitive theory in terms of a computer program, while perfectly rigorous, is not necessarily perspicuous. For example, in the current essay the use of the ATN helps us to organize the procedural knowledge we are trying to characterize. A second theme is that problem solving consists primarily of two complementary activities: *planning* and *debugging*. Previous research has typically emphasized only one or the other, at the expense of both. A goal of our theory is to provide an integrated understanding of both processes. A final theme is that detailed study of the problem solving involved in program design is a prerequisite for completely fulfilling the *structured programming* movement's objectives, such as program reliability. We wish to emphasize the potential role of our research in this enterprise.

2. See also [Woods, Kaplan & Webber 1972]. Woods' [1970] definition was an elaboration and formalization of earlier work by Bobrow and Fraser [1969], and by Thorne, Bratley and Dewar [1968]. Woods attributes some aspects of the ideas to Kuno [1965] and Conway [1963].

3. While the emphasis of the current essay is on investigating the appropriateness of an ATN formalism for planning concepts, we have also found the context free grammar representation to be a fruitful description of planning concepts for certain purposes, such as parsing human programming protocols. This suggests that Heidorn's [1975] *ACFG* (*augmented context free grammar*) formalism might be an effective alternative to the ATN. Its virtue is that the relationship to the CFG characterization of our ideas would be more direct. Moreover, our actual implementation of PATN might turn out to be closer in spirit to an ACFG model than an ATN. To some extent, the distinction is secondary, since ACFG's and ATN's are not only formally equivalent in power, but also structurally comparable in a straightforward manner. In any case, while ACFG's suggest interesting possibilities, resolution of this issue goes beyond the current paper.

4. We should emphasize that we do not regard this taxonomy as being either complete or unique. In later sections we discuss particular ways in which it is incomplete. In [Miller & Goldstein 1976b] we presented a different version, in the context of parsing a student protocol. The earlier taxonomy emphasized examining the directions from whence a planner could obtain guidance; the current one emphasizes examining the logistic description of the problem at hand. While our intuition suggests that our current version is an improvement, persuasive evidence for favoring a given classification of planning concepts must await implementation and systematic experimentation. The reader is referred to [Miller & Goldstein 1976a] for an overview of our research project as a whole.

5. This is an oversimplification. If every solved problem were added to the answer library, the experienced problem solver might be overwhelmed by tremendous numbers of uninteresting solutions. The possibility of "intelligent forgetting" is a subtle issue which we are not currently in a position to address.

6. Our use of the term *model* should not be confused with its use in model theory. The name clash is unfortunate, resulting from historical accident. In most cases our term *model* can be replaced by the phrase *problem specification* without altering the meaning.

7. The predicate calculus is the problem description language of mathematics as well as a variety of AI programs, most notably the STRIPS series of problem solvers [Fikes & Nilsson 1971; Fikes 1972; Fikes et al. 1972]. Alternative problem description languages, based upon such concepts as frames [Minsky 1975; Winograd 1975; Goldstein 1975], might provide increased expressive power; we have yet to thoroughly explore this issue. For our purposes in this article, the problem descriptions are simply a conjunction of properties and relations about some set of objects. As such, they are common to most descriptive schemes including the predicate calculus, frames, and semantic nets [Quillian, 1968; Winston 1975; Woods 1975]. In practice, of course, our problem specification language is actually LISP: but the subset of LISP which is used can be viewed in a variety of guises.

8. It is possible that problems should also be indexed by their Pre Models, if any. This would enable the system to support a kind of *forward chaining*. At the present time, the additional overhead which this would entail does not seem justified by its possible utility, at least for the simple blocks world and Logo picture problems we are considering.

9. For a more detailed discussion of the link between turtle primitives and model descriptions, see chapter six of [Goldstein 1974]. A glossary of primitive predicates for describing elementary Logo pictures may be found in Goldstein's Appendix B.

10. Only implementation and experimentation can ultimately determine whether a given set of reformulation techniques will be adequate. A related problem for future research is to construct a program that attempts to induce the model from a sketch. The potential ambiguity introduced by such a module would place an even greater burden on the reformulation strategies.

11. See [Polya 1965, ch. 9] for a relevant discussion of Problems within Problems.

12. There are of course many additional reformulation techniques. Many complex issues involving change of representation arise, suggesting rich areas for further research.

13. Strictly speaking, what Sacerdoti terms *procedural nets* are actually *partially ordered program steps*. The authors are indebted to B. Kuipers for reminding them that such partial orders are restricted cases of networks, with additional properties useful to both NOAH and PATN.

14. An alternative is to save the solutions to subproblems only in a working *lemma library*. The issue is whether each lemma is permanently stored for future reference, or only saved for the duration of the problem at hand. Techniques for determining the potential future relevance of subproblems are not discussed in this paper.

15. We introduce DAPR here because we have found that the metaphor of *designing a program* is a useful way to organize our ideas. We do, in fact, intend for this design to serve as the basis for implementing a debugging module. At the same time, we are aware that the set of ideas presented are incomplete: the architecture of the debugging module, DAPR, is only partially specified in this report.

16. This view of the causes of bugs is a simplification. Some bugs have multiple underlying causes, a situation which greatly complicates the troubleshooting process. Nevertheless, the techniques developed here are useful, in that proceeding under the heuristic assumption of a single cause is often reasonable even in cases where the assumption turns out to be false.

17. In the general case, model diagnosis requires addressing difficult problems of symbolic evaluation (see, e.g., [Yonezawa 1976]). For most of the programs discussed here, a simpler approach, *performance annotation* [Goldstein 1974] is possible. A direction for research is to extend the range of programming constructs which can be verified by the model diagnosis module.

18. DAPR's three diagnostic techniques are presented in roughly the order in which they would be applied. It is conceivable that this strict ordering would not be adequate. That is, there may be debugging situations for which process diagnosis should be applied prior to model diagnosis, or even situations for which the most effective debugging strategy would be alternate applications of both strategies, and so on. In the first implementation of DAPR we will experiment with the simpler approach.

19. *Careful evaluation* [Hewitt & Smith 1975; Goldstein 1974; Sussman 1973] is a diagnostic tool whereby a program is tested by interpreting it in an extremely cautious mode, with extensive checking of argument types, prerequisite satisfaction, etc. During normal evaluation it would be prohibitively expensive to routinely include such checks.

20. PATN's default arc ordering and arc constraints are designed to ensure that non-linear planning is pursued if and only if a specific pattern of interaction is detected. The local decision process may be described as follows. PATN first tries the two nonlinear arcs. Control transfers to the corresponding states only when an NLC or NLD predicate "accepts" the model. Otherwise, the linear decomposition is pursued.

21. It is worth considering the relationship between Newell & Simon's [1972] *production system* model and PATN. Strictly, ATNs are isomorphic to production systems in formal power; they are also directly analogous in internal structure. A production system consists of a set of [pattern => action] rules which operate over a finite number of *short term memory (STM)* locations. An ATN may be thought of as a production system in which a particular slot in STM, the *state*, is distinguished. The arc transitions correspond to rules, where arc contraints map onto the left hand sides, and arc actions map onto the right hand sides. Distinguishing between the "state" register and other ("data") registers seems to have the virtue of imposing greater structure on the otherwise homogeneous collection of productions. All the reputed advantages of rule-based systems, such as modularity, still apply. The other STM slots directly correspond to the registers of the ATN model. Moreover, the ATN model suggests a natural decomposition of the knowledge in a given rule, into syntax, semantics, and pragmatic constraints. One application of this breakdown is in teaching: rather than tutoring an entire rule, it may be that only one part need be taught. (The authors are indebted to B. Kuipers for emphasizing the importance of this comparison.)

8. References

[Aho and Ullman 1972]
Aho, A.V., and J.D. Ullman. *The Theory of Parsing, translation, and Compiling* (Volume I: Parsing), Prentice-Hall, Englewood Cliffs, N.J. 1972.

[Allen 1975]
Allen, James F. "A Speech Understanding System Based Upon a Co-routine Parser," *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, Tbilisi, Georgia, USSR, September 3-8 1975, pp. 455-460.

[Bobrow and Fraser 1969]
Bobrow, D.G., and J.B. Fraser. "An Augmented State Transition Network Analysis Procedure," *Proc. Internat. Joint Conf. on Artificial Intelligence*, Washington D.C. 1969, pp. 557-567.

[Burton & Brown 1976]
Burton, Richard R., and John Seely Brown, "A Tutoring and Student Modelling Paradigm for Gaming Environments," in R. Colman and P. Lorton Jr. (eds.), *Computer Science and Education* (Advance Proceedings of the Association for Computing Machinery Special Interest Groups on Computer Science Education and Computer Uses in Education Joint Symposium, Anaheim, Cal.), SIGCSE Bulletin, Volume 8, Number 1 (SIGCUE Topics Volume 2), February 1976, pp. 236-246.

[Conway 1963]
Conway, M.E. "Design of a Separable Transition-Diagram Compiler," *Communications of the ACM*, Vol. 6, No. 7 (July 1963), 396-408.

[Dahl et al. 1972]
Dahl, Ole-Johan, Edsger Dijkstra and C.A.R. Hoare, *Structured Programming*, London, Academic Press, 1972.

[Fahlman 1974]
Fahlman, Scott, "A Planning System for Robot Construction Tasks," in *Artificial Intelligence*, vol. 5, 1974, pp. 1-49.

[Fikes 1972]
Fikes, Richard E., "Monitored Execution of Robot Plans Produced by STRIPS," in *Information Processing*, Vol. 71, 1972.

[Fikes & Nilsson 1971]
Fikes, Richard E. and Nils J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," in *Artificial Intelligence*, Vol. 2, 1971, pp. 189-208.

[Fikes et al. 1972]
Fikes, Richard E., Peter E. Hart and Nils J. Nilsson, "Learning and Executing Generalized Robot Plans," in *Artificial Intelligence*, Vol. 3, 1972, pp. 251-288.

[G. Goldstein 1973]
Goldstein, Gerrianne, *LOGO Classes Commentary*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, LOGO Working Paper 5, February, 1973.

[Goldstein 1974]
Goldstein, Ira P., "Understanding Simple Picture Programs," in *Artificial Intelligence*, Vol. 6, No. 3, 1975; and Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Technical Report 294, September 1974.

[Goldstein 1975]
Goldstein, Ira P., "Bargaining Between Goals," in *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, Tbilisi, Georgia, USSR, September 1975, pp. 175-180.

[Goldstein & Miller 1976a]
Goldstein, Ira P., and Mark L. Miller. *AI Based Personal Learning Environments: Directions for Long Term Research*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 384 (LOGO Memo 31), December 1976a.

[Heidorn 1975]
Heidorn, George E. "Augmented Phrase Structure Grammars," *Theoretical Issues in Natural Language Processing*, Cambridge, Mass., Association for Computational Linguistics, June 1975.

[Hewitt 1972]
Hewitt, Carl, *Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Technical Report 258, April 1972.

[Hewitt & Smith 1975]
Hewitt, Carl, and Brian Smith. "Towards a Programming Apprentice," *IEEE Transactions on Software Engineering*, Volume SE-1, Number 1, March 1975.

[Kaplan 1973]
Kaplan, R. "A General Syntactic Processor," in R. Rustin (ed.), *Natural Language Processing*, Algorithmics Press, New York, 1973.

[Kay 1973]
Kay, Martin, "The MIND System," in Randall Rustin (ed.), *Natural Language Processing*, Courant Computer Science Symposium 8 (December 20-21, 1971), New York, Algorithmics Press, 1973, pp. 155-188.

[Kowalski 1973]
Kowalski, Robert, *Predicate Logic as a Programming Language*, University of Edinburgh, Department of Computational Logic, School of Artificial Intelligence, Memo 70, 1973.

[Kuno 1965]
Kuno, S. "A System for Transformational Analysis," In *Report NSF-15*, Computer Laboratory, Harvard University., Cambridge, Mass., 1965.

[Kuno 1967]
Kuno, S. "Computer Analysis of Natural Languages," *Proceedings of Symposia in Applied Mathematics*, Volume 19, American Mathematical Society, 1967.

[Lesser et al. 1975]
Lesser, V.R., R.D. Fennell, L.D. Erman and D.R. Reddy, "Organization of the Hearsay II Speech Understanding System," in *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. Assp-23, No. 1, February 1975, pp. 11-24.

[Miller & Goldstein 1976a]
Miller, Mark L., and Ira P. Goldstein. *Overview of a Linguistic Theory of Design*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 383 (LOGO Memo 30), December 1976a.

[Miller & Goldstein 1976b]
Miller, Mark L., and Ira P. Goldstein. *Parsing Protocols Using Problem Solving Grammars*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 385 (LOGO Memo 32), December 1976b.

[Miller & Goldstein 1976c]
Miller, Mark L., and Ira P. Goldstein. *SPADE: A Grammar Based Editor for Planning and Debugging Programs*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 386 (LOGO Memo 33), December 1976c.

[Miller & Goldstein 1976d]
Miller, Mark L., and Ira P. Goldstein. *PAZATN: A Linguistic Approach to Automatic Analysis of Elementary Programming Protocols*, Artificial Intelligence Laboratory, Memo 388 (LOGO Memo 35), December 1976d.

[Mills 1973]
Mills, Harlan D. "On the Development of Large Reliable Programs," *IEEE Symposium on Computer Software Reliability*, New York, April 1973, pp. 155-159.

[Minsky 1975]
Minsky, Marvin, "A Framework for Representing Knowledge," in P. Winston (ed.), *The Psychology of Computer Vision*, New York, McGraw-Hill, 1975, pp. 211-277.

[Minsky & Papert 1974]
Minsky, Marvin and Seymour Papert, *Artificial Intelligence*, Condon Lectures, Oregon State System of Higher Education, 1974.

[Moore 1975]
Moore, Robert, *Reasoning from Incomplete Knowledge in a Procedural Deductive System*, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Technical Report 347, December 1975.

[Newell & Simon 1972]
Newell, Allen and Herbert Simon (eds.), *Human Problem Solving*, Englewood Cliffs, New Jersey, Prentice-Hall, Inc., 1972.

[Okumura 1973]
Okumura, K., *LOGO Classes Commentary*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, LOGO Working Paper 6, February 1973.

[Papert 1971a]
Papert, Seymour A., *Teaching Children to be Mathematicians Versus Teaching About Mathematics*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 249, 1971.

[Papert 1971b]
Papert, Seymour A., *Teaching Children Thinking*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 247 (LOGO Memo 2), 1971.

[Papert 1973]
Papert, Seymour A., *Uses of Technology to Enhance Education*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 298 (LOGO Memo 8), June 1973.

[Paxton & Robinson 1975]
Paxton, William and Ann Robinson, "System Integration and Control in a Speech Understanding System," in *American Journal of Computational Linguistics*, Vol. 5, 1975, pp. 5-18.

[Polya 1957]
Polya, George, *How To Solve It*, New York, Doubleday Anchor Books, 1957.

[Polya 1962]
Polya, George, *Mathematical Discovery* (Volume I), New York, John Wiley and Sons, 1962.

[Polya 1965]
Polya, George, *Mathematical Discovery* (Volume 2), New York, John Wiley and Sons, 1965.

[Polya 1967]
Polya, George, *Mathematics and Plausible Reasoning* (Volumes I & 2), New Jersey, Princeton University Press, 1967&8.

[Quillian 1968]
Quillian, M. Ross, "Semantic Memory," in M. Minsky (ed.), *Semantic Information Processing*, Cambridge, Massachusetts, The MIT Press, 1968.

[Rapport 1963]
Rapport, Samuel and Helen Wright (eds.), *Mathematics*, New York University Press, 1963.

[Rubin 1975]
Rubin, Andee, *Hypothesis Formation and Evaluation in Medical Diagnosis*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Technical Report 316, January 1975.

[Sacerdoti 1975]
Sacerdoti, Earl, "The Nonlinear Nature of Plans," in *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, Tbilisi, Georgia, USSR, September 1975, pp. 206-218.

[Simon 1969]
Simon, Herbert A. *The Sciences of the Artificial*, Cambridge, Mass., MIT Press, 1969.

[Sussman 1973]
Sussman, Gerald Jay, *A Computational Model of Skill Acquisition*, New York, American Elsevier, 1975; and Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Technical Report 297, 1973.

[Sussman 1974]
Sussman, Gerald Jay, "The Virtuous Nature of Bugs," *AISB Summer Conference*, July 1974, pp. 224-237.

[Thorne et al. 1968]
Thorne, J., P. Bratley and H. Dewar, "The Syntactic Analysis of English by Machine," in D. Michie (ed.), *Machine Intelligence 3*, New York, American Elsevier, 1968.

[Winograd 1972]
Winograd, Terry, *Understanding Natural Language*, New York, Academic Press, 1972.

[Winograd 1975]
Winograd, Terry, "Frame Representations and the Declarative-Procedural Controversy," in D. Bobrow and A. Collins (eds.), *Representation and Understanding: Studies in Cognitive Science*, New York, Academic Press, 1975, pp. 185-210.

[Winston 1975]
Winston, Patrick, "Learning Structural Descriptions from Examples," in Patrick Winston (ed.), *The Psychology of Computer Vision*, New York, McGraw-Hill, 1975, pp. 157-209.

[Woods 1970]
Woods, William A., "Transition Network Grammars for Natural Language Analysis," *Communications of the ACM*, Vol. 13, No. 10, October, 1970, pp. 591-606.

[Woods 1975]
Woods, William A., "What's in a Link: Foundations for Semantic Networks," in D. Bobrow and A. Collins (eds.), *Representation and Understanding: Studies in Cognitive Science*, New York, Academic Press, 1975, pp. 35-81.

[Woods et al. 1975]
Woods, William A., Madeleine Bates, Geoffrey Brown, Bertram Bruce, John W. Klovstad and Bonnie Nash-Webber, *Uses of Higher Level Knowledge in a Speech Understanding System*, Bolt, Beranek and Newman, Report 3240, December 1975.

[Woods et al. 1972]
Woods, William A., R.M. Kaplan and Bonnie Nash-Webber, *The Lunar Sciences Natural Language Information System* (Final Report), Bolt, Beranek and Newman, Report 2378, 1972.

[Yonezawa 1976]
Yonezawa, Akinori. *Symbolic-Evaluation as an Aid to Program Synthesis*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Working Paper 124, April 1976.